An introduction to

DIGITAL ELECTRONICS

4/-

THIRD EDITION — PRICE $3.50

AUTO TAPE PLAYERS BALUNS
HOLDERS & CLIPS BEZELS BOARDS
BURGLAR ALARM PARTS BUZZERS CB
SSORIES CAR AERIALS CAR ALARMS CAR RADIOS
RTRIDGES CASES CASSETTE DECKS CASSETTE PLAYERS
CLOCKS CLOCK RADIOS COAX
RECIEVERS COMMUNICATION INSTRUMENTS CONDENSE
COUPLES CROCODILE CLIPS CROSSOVERS CRYS
DRIVERS SPRINGS DIGITAL VOLTMETERDIMMER DIODES DIRECTIO
DRILL SPEED CONTROL DUMMMY DUST BUGS HOOK EAR PHONE
TIC CAPACITORS ENGINE ENGRAVING
RIEZS FILTERS LAMPS FLUX FREEZE NCYCOUNTERS
GDO GLOBES GLUE GRILL CLOTH GRILLES GUITAR PICK
HEADPHONES HEAT SHINK TUBING HI FI SPEAKERS
ANSFORMERS IGNITION MENTS INSULA
UITS INTERCOMS LAMP HOLDER
LED'S LETTERING UGS MAGNETS MA
AD AMP MATRIX BOARD MICROPHONES MIXE
MOTOR ACCES COLOR NEON LAMP
RGAN OSCILLOSCOPE LIGHT PLUGS SOCKETS PO
ETERS POWER SUPP BOARDS PRINTED CIRCUIT
IC ADDRESS EQUIP R MODULE RADIO AUTOMO
VERS COMMUNIC TIFIERS RELAYS RESISTOR
STOR MOUNTI CHOKES RHYTHM UNITS SCR
CONDUCTORS SILICON GREASE SIRENS
SOCKETS SOL SPACERS SPAGHETTI SPEAKER BOXE
SPEAKERS STEREO CARTRIDGES STRIP BOARDS STROBE LIG
STYLUS GAU SWR METERS TAG STRIPS TAPE ACCESSORIES TAPE RE
TELESCOP TERMINALS TEST EQUIPMENT TEST LEADS
TOOLS TOROIDS TRANS TRANSFERS TRANSISTORS TRANSIST
TRIGGER TRANSFORMERS TUBES ANTENNAS TV ACCESSO
TWO WAY RADIOS ULTRASONIC TRANSDUCERS UTILUX VALVES VALVE
VIBROSCOPE VIDEO GAMES VOLUME CONTROLS WAH WAH
ZENER DIODES ACOUSTIC PADDING ADAPTORS ALARMS
AMPLIFIERS ANTENNA MATCHERS
ACOUSTIC PADDING ADAPTOR ALARMS
AUTO ELECTRONI ALARM PARTS
SPEED CONTROL DUMMMY
FLURO LAMPS
GRILL CLOTH GUITAR PICK
HEADPHONES HI FI SP
INTERCOMS KNOBS LAM
LETTERING DIMMER LIGHT SHOWS TRAINER LUGS
MATRIX BOARD METAL DETECTOR METER COMPUTER MICRO
ACCESSORIES EQUIPMENT MUSICOL
PILOT LIGHT PL
CIRCUIT BOARDS
RADAR MODU
RECTIFIER
RESISTORS RF CHOK
SIGNAL GENER IZERS SI
SPACERS
RTRIDGES
TAG STRIPS
TEST EQU
FORMERS
TURNT
SONIC TRA UTILUX
VOLUME CONTROLS WAH
ACOUSTIC PADDING

LETS FACE IT...

You only need to make one stop for all your electronic and hobby needs . . .

DSE36

DICK SMITH ELECTRONICS

bankcard welcome here

SYDNEY
125 York Street, SYDNEY. Ph. 29 1126
147 Hume Hwy, CHULLORA. Ph 642 8922
162 Pacific Hwy, GORE HILL. Ph 439 5311
30 Grose Street, PARRAMATTA Ph 683 1133

MELBOURNE 399 Lonsdale Street, MELBOURNE. Ph 67 9834
656 Bridge Road, RICHMOND. Ph 42 1614

BRISBANE 166 Logan Road, BURANDA. Ph 391 6233

ADELAIDE 203 Wright Street, ADELAIDE. Ph 212 1962

MAIL ORDERS P.O. Box 747, Crows Nest, N.S.W. 2065.

SHOPS OPEN 9AM to 5.30PM
(Saturday 9AM 12 noon)
BRISBANE 1/2 Hour earlier

Some lines may be available from the Dick Smith Electronics Centres at

GRACE BROS

Broadway • Bondi Junction • Chatswood
Liverpool • Miranda Fair • Mt Druitt •
Parramatta • Roselands • Warringah Mall

®

An Introduction to

Digital Electronics

by JAMIESON ROWE

BA (Sydney), BSc (Technology, NSW), MIREE (Aust.)
Editor, "Electronics Australia"

Third Edition, 1978 — updated & revised

Phototypeset by Dalley-Middleton-Moore Pty Ltd of Wattle St, Sydney, and Magazine Printers Pty Ltd of Regent St, Sydney, and printed by Masterprint Pty Ltd of Dubbo, NSW, for Sungravure Pty Ltd of Regent St, Sydney.

Preface

Back in 1966-67, I wrote a series of articles which were published in Electronics Australia under the title "Logic and Counting Circuits". The articles were written with the idea of helping readers to become familiar with the concepts of logic, which were just beginning to appear in electronics, and with the operation of instruments like frequency counters and digital voltmeters which were then also making their appearance.

Even as the series was being written, it became apparent that the concepts involved were becoming more and more important for electronics as a whole. The original scope of the articles was therefore extended, to cover at least in basic form some of the previously esoteric concepts involved in digital computers.

The response to the resulting articles was very warm, and we were prompted to republish them in handbook form. When this was done we gave them a new title more in keeping with the expanded scope: An Introduction to Digital Electronics.

Like the articles from which it was produced, the original book was very well received. It sold tens of thousands, and became widely used not only by private individuals, but by many schools, technical colleges and universities. A second and revised edition was produced, and, like the first, it was reprinted a number of times.

However, as time has passed and digital electronics has grown in importance so dramatically, it has become increasingly obvious that a fully updated and revised book is required: one which deals with the many devices, circuits and techniques which have evolved since the original material was written. Devices like LSI microprocessors, RAM, ROM and PROM memory devices, calculator chips and UARTs, circuits for video character generation, priority encoding and code translation; and techniques like multiplexing, synchronous data transmission and microprogramming.

The material in this present book has been written in an attempt to meet these needs. It is not simply a revision or updated version of the original book; in fact if you compare the two, you will find that they are quite different. I found it necessary to start again from scratch, in order to give the new book a sufficiently broad and modern outlook. This has also provided the opportunity to re-express some of the basic concepts, and I believe they are now expressed rather more clearly than before.

Unfortunately to cover the whole scope of digital electronics now requires a book much larger than the original — and to write such a book requires considerable time and effort. It seems unwise to delay publication of much of the basic material until all of the more specialised and applied material can be written and published. Accordingly we have taken the step of publishing the present book, a transition volume covering the 19 chapters which have been generated to date.

Further material will be published as soon as possible, initially on a chapter-by-chapter basis in "Electronics Australia". In the meantime, I hope these introductory chapters help you understand the basic concepts involved in the exciting and ever-growing world of digital electronics. If so my efforts will have been worthwhile.

Needless to say no work of this type is the work of a single individual, but rather a team effort. With that in mind I would like to express thanks to draftsman Bob Flynn for his excellent work on the many diagrams, and the other EA staff members for their constructive criticism and other assistance.

Jamieson Rowe,
20th September, 1978

Contents

Main Chapters

Chapter 1

Signals, circuits and logic

One of the things which puzzles and confuses many people when they first come across digital circuits and systems is the many terms and ideas derived from fields apparently quite unrelated to electronics. In this introduction the author explains how it is that things like Boolean algebra, logic, number systems and computers have become an integral part of digital electronics.

In the broadest sense, electronics is concerned with physically manipulating an abstract quantity—information. This rather ingenious feat is done by using current or voltage changes to represent the information, often in encoded form. The current or voltage changes are given the general label of "signals".

There are essentially two quite different ways of representing information as electrical signals, to allow us to manipulate it. One way is to make an electrical current or voltage vary continuously, in a manner which directly copies the information itself. Because the current or voltage variations are effectively the electrical analogy of the original information, this technique is known as the "analog" approach.

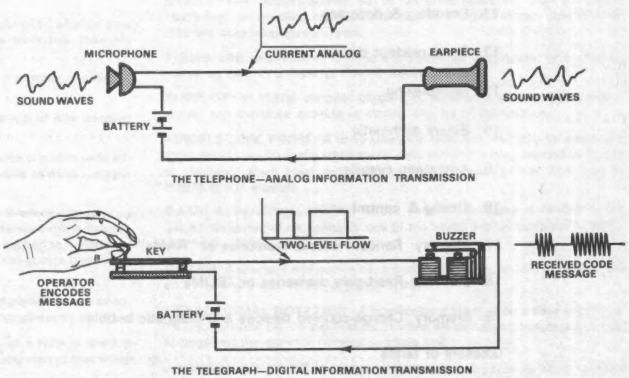
Conventional radio and TV broadcasting both use the analog method, and so does sound recording. For example in a recording studio, a microphone is used to produce a very small varying current, which copies the sound pressure variations in the studio. This small signal is then amplified, to produce a larger varying current with substantially the same variations, and used to make a recording—say in the form of variations in the magnetisation of a roll of recording tape. Upon playback, a magnetic head generates a small electrical signal once more, and after amplification the electric signal is fed to a loudspeaker to recreate a very close approximation to the original sound information.

Although this analog approach works quite well, it is not without its problems. One major problem is that in order to handle analog signals faithfully, electrical circuits must behave in a very "linear" fashion—that is, their output should be directly proportional to their input. Perfect linearity is impossible to achieve, however, and as a result signals in analog form tend to become "distorted" as they progress through circuits.

Another problem is noise. All electrical circuits tend to generate small random current variations, known as noise. This inevitably tends to add to analog signals passing through the circuits, and because there is often no easy way of distinguishing noise variations from the "true" signal variations, the signals are effectively degraded.

The alternative approach used to represent information in electrical signal form avoids these problems. Instead of using the electrical signal as a direct analogy of the information, this method encodes the information and uses the electrical signal merely as a vehicle to convey the resulting code. The electrical current or voltage is not made to vary continuously, but is switched between a relatively small number of distinct levels.

Most frequently there are only two levels, one of which may correspond to no current (or no voltage). In other words, the two levels may be "current" and "no current", with the encoded information represented by various sequences of these two levels.



A simple illustration of the two ways of manipulating information electronically—the analog method and the digital method.

This method of representing and manipulating information in electrical form is known as the digital approach, mainly because the encoded information may often be visualised as a series of numerical digits.

The digital approach may seem less familiar than the analog method at first hand, but it has really been in use for a very long time. Most early signalling systems were digital; from the smoke signals of cave men, through fire towers and mechanical semaphore systems, to the early electrical telegraph systems based on Morse code.

In fact the electrical telephone and telegraph are very simple examples which illustrate the difference between the analog and digital methods. In the telephone, the information is transmitted from one end to the other via a current which varies continuously as a direct equivalent of the sound waves striking the microphone—the analog approach. In the telegraph, the information is encoded and sent as a sequence of current/no current pulses, illustrating the digital approach.

In comparison with the analog approach, the digital method of representing and manipulating information tends to be less demanding in terms of electrical circuit performance. It does not require circuits to be "linear", merely requiring that they switch between two or more fixed and well-defined states—often simply "on" and "off", or "high" and "low". This means that distortion tends to be less of a problem.

Similarly because the levels used tend to be quite distinct, noise also tends to be less of a problem. For example with a digital system using only two levels—say "on" and "off"—any noise introduced into the system has to become very large indeed before it is capable of degrading the wanted information. In fact it has to become large enough to make the "on" level capable of being mistaken for "off", and vice-versa.

Because of these advantages, digital techniques have become very widely used in electronics, and they are likely to become even more common in the future. It is already true to say that the digital approach has been used to advantage in almost every field of electronics—even in sound and television recording and broadcasting, which have been traditional analog strongholds.

An understanding of the basic principles involved in digital electronics is therefore going to be very necessary for anyone aiming to be involved with electronics in the future. Our aim here will be to provide you with this understanding.

Historically, the techniques of digital electronics grew from electrical telegraphy and from early work on switching and control circuits. Along the way, the people who developed the techniques found it worthwhile to borrow ideas and concepts from many other fields, some of which may seem at first sight to be quite unrelated to traditional electronics.

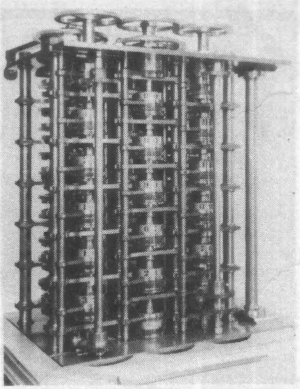
One of these fields was Boolean algebra, developed around 1850 by the logician George Boole as a way of representing and analysing the relationships between classes or sets of objects. The work of Boole and his contemporary Augustus de Morgan was regarded as rather abstract and theoretical at the time, but when digital electronics was starting to appear in the 1930's, the brilliant US engineer Claude Shannon discovered that the concepts Boole and de Morgan had developed were ideally suited for the design of digital circuits.

A key concept of Boolean algebra was that the logical relationships between classes, no matter how complex, could always be broken down into combinations of a relatively small number of basic or fundamental relationships. These "logical operators" were thus seen as the building blocks of all complex logical relationships.

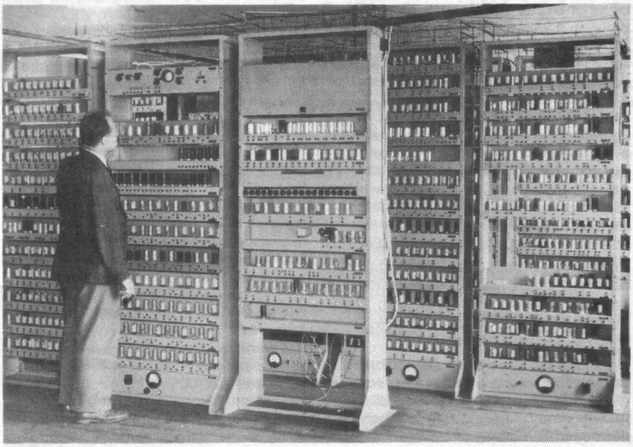
Shannon discovered that the same concepts also applied to digital circuit design. If a circuit was required to perform a relatively complex function, it could be built up efficiently and elegantly using circuit "building blocks", based directly upon the fundamental logic operators of Boolean algebra. Naturally this discovery took a lot of the guesswork and trial-and-error out of the design of digital circuits, and made it possible to design much more complex circuits.

We will look more closely at the fundamental logic operators in the next chapter, as they are important enough to deserve a full discussion on their own. But perhaps you can see already why the concepts of logic start cropping up as soon as you look into books and articles on digital electronics. And also the reason why digital circuits are often alternatively called "logic" circuits.

Another field whose ideas became incorporated into digital electronics was the branch of mathematics concerned with numerical notation and number systems. In particular it was found that a number system ideally suited for use in digital circuits was "binary" notation, based on powers of two. This had been discovered early in the 17th century, by either Sir Francis Bacon or the Scottish mathematician John Napier (inventor of logarithms).



Charles Babbage, gifted mathematician of the early 19th century, and his "analytical engine"—an early mechanical digital computer.



EDSAC, one of the first electronic digital computers, built at Cambridge University in 1949. The shot at right shows Dr. M. V. Wilkes, leader of the team which produced the machine. It used more than 3000 thermionic valves. The name "EDSAC" stood for electronic delay storage automatic calculator.

Just a few of the many applications of digital electronics. At upper left is a minicomputer on a single PC board, both "naked" and in a case (Computer Automation); at upper right, a computerised drafting table (IBM, Gerber); at left, a programmable pocket calculator (Hewlett-Packard); above centre, an electronic clock (Digeec); above right, a digital multimeter (Non-Linear Systems).

Unlike the familiar decimal system based on powers of ten, the binary system requires only two different digit symbols—0 and 1. Numbers are represented by combinations of the two, using positional order to indicate the various powers of two. It is therefore very easy for numbers in binary form to be represented by the two current or voltage levels of most digital circuitry. Thus "no current" may be used to represent 0 and "current" to represent 1, or vice-versa.

Binary notation and binary arithmetic are in fact used very widely in digital circuits, and will be discussed in some detail in later chapters.

Another activity which not only influenced, but virtually became intertwined with digital electronics was the development of calculating and computing machines. Men had long dreamed of having machines to free them from the drudgery of repetitive and lengthy calculations, and as early as 1650 crude machines were developed.

Among the more notable were Blaise Pascal's adding machine, Gottfried von Leibnitz's four-function calculator, and Charles Babbage's "difference" and "analytical" engines.

But it was only in the 1930's and forties that really practical computing and calculating machines were developed, using some of the ideas of digital electronics, and in turn helping to expand the concepts of digital electronics in new directions. In 1944 a team led by Dr Howard Aitken at Harvard University produced a relay calculator, while two years later John Mauchly and J. Presper Eckert of the University of Pennsylvania produced "ENIAC", the first wired-program digital computer. In 1949 the first true stored-program general purpose digital computers appeared, developed almost simultaneously at Cambridge University, Manchester University and Princeton Institute of Advanced Study.

Since that time, of course, digital computers have developed in leaps and bounds, and digital electronics has developed largely in parallel with them. Further discussion of computers will also be given in later chapters.

Yet another seemingly unrelated activity from which digital electronics borrowed ideas was weaving. In 1801 the weaving loom maker Joseph Jacquard developed a method of "programming" a loom to automatically weave complex patterns, using cards in which holes were punched. The cards were strung together and fed through the loom in sequence, with sensing needles setting the loom threads according to the "instruction code" formed by the holes.

The idea became used very widely in the weaving industry, but it was not until 1889 that the US statistician Dr Herman Hollerith hit upon the idea of using punched cards to store information for calculating and manipulation. Subsequently, punched cards and punched paper tape played a very important role in the practical development of computers, telegraphy and digital control of machinery.

These and many other concepts from all sorts of other fields have gone together to produce the widely-embracing activity now known as digital electronics. And as you are probably already aware, the resulting techniques have application not only in the more dramatic fields of computers and automation, but also in areas like communication, broadcasting, sound and vision recording, electronic musical instruments and synthesisers, banking, cash terminals, medical instrumentation, games of skill and chance, printing, measurement and timekeeping. Many of these applications will be discussed in more detail as we progress through the subject.

Chapter 2

Basic logic elements

When designing digital circuits, and also when analysing their operation, it is very helpful to think in terms of logic function. To do this effectively one needs to have a clear understanding of the fundamental logic functions AND, OR and NOT, together with their derived functions NAND, NOR and exclusive-OR.

As we noted in the first chapter, Claude Shannon showed that the concepts developed by George Boole and Augustus de Morgan for dealing with class relationships could be used almost without change to analyse and design digital circuits. An important implication of this is that no matter how complex a digital "logic" circuit may be, it is always possible to break it down into a combination of certain fundamental logic operations. These fundamental logic operations or "functions" are effectively the building blocks from which digital circuits and systems are assembled.

There are really only three of these fundamental logic functions, known by the labels AND, OR and NOT. Although some fairly simple combinations of these are also used as basic building blocks, AND, OR and NOT are the only really fundamental logic functions.

Because they are so basic, the three functions are not very hard to understand once you get used to the concepts involved. But, because they play such an important role in all digital circuits and systems, it is essential that you do really understand them right from the start—otherwise the rest of digital electronics will never really make much sense. So we are now going to look at the three functions in some detail.

Basically the logic label AND is used to describe any situation where one event or state-of-affairs occurs only if there is a combination or "conjunction" of certain other events or states of affairs. Putting it another way, any situation where something happens only when certain conditions are met simultaneously can be described as an example of the AND function in operation.

Let's look at a simple example. In the lamp switching circuit of Fig. 1, it should be fairly obvious that current will only flow through the lamp if all three switches are closed at the same time. It is therefore true to say that the relationship between the lamp being lit and the three switches being closed is an example of the logical AND operation.

Now look at Fig. 2, which is again a very elementary switching circuit with three switches and a lamp. Here the three switches are in parallel, so that if any one of the switches is closed, current will flow and the lamp will light. The relationship between the lamp being lit and the three switches being closed is now obviously different from that in Fig. 1. In fact, it corresponds to the second fundamental logic function, the OR operation.

The logical label OR is used to describe any situation where the occurrence of one event or state-of-affairs depends on the occurrence of any one, but at least one, of a group of other events or states-of-affairs. In other words, the OR function deals with situations where all that is necessary for something to occur is that at least one of a group of conditions has occurred.

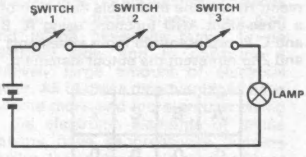


FIG. 1

Note that the OR function describes a situation where a number of possibilities can result in the controlled event occurring: either only one of the conditions can apply, or two, or any number up to and including the case where ALL conditions apply. In a sense, therefore, the OR function "includes" the AND function—or, more strictly, it covers situations where the single set of conditions defined for the AND function is one of a number of possibilities, any of which is sufficient for the controlled event to occur.

Another important thing to notice is that, in another sense, the AND and OR operations are really the opposite sides of the same coin. Look again at Fig. 1, but this time visualise the controlled event not as being the flow of current and lighting of the lamp, but as the interruption of current and extinguishing of the lamp. It should now be apparent that, instead of illustrating the AND function, it will illustrate the OR function. This is because the lamp will extinguish if any one or more of the switches is opened.

Now look at Fig. 2, and again visualise the controlled event as being the extinguishing of the lamp. Fairly obviously, when considered from this angle, the circuit is now illustrating the AND function instead of the OR function. All three of the switches must now be open before the lamp will extinguish.

In other words the operation of the circuit in Fig. 1 is equally capable of representing either the AND or OR functions, depending upon the way we choose to look at it. And the same applies to the circuit in Fig. 2, the only difference being that the relationship between the two logical functions and the electrical events is reversed.

You can see from this, I hope, the way in which AND and OR are in effect only the opposite sides of the same logical coin. This is in fact a very important concept; it forms the basis of the Boolean algebra law known as "de Morgan's theorem", as we shall see later on.

The other point which hopefully, has also become apparent is that the relationship between an electrical circuit and the logical function it is seen to perform is not fixed, but capable of being arbitrarily assigned. In other words the circuit in Fig. 1 could be used to perform either an AND or OR function at will, simply by selecting the appropriate relationships between the electrical events and the logical information "quantities" they are used to represent.

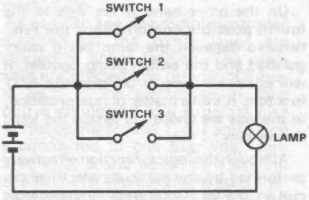


FIG. 2

This is a very important concept to grasp, because it means that physical circuit modules used as logic elements are never rigidly fixed in terms of their logic function. A module can be used to perform a variety of functions as desired, simply by adopting the appropriate interpretation of its electrical behaviour.

Selecting the appropriate logical interpretation of a circuit's electrical behaviour is usually called "assigning the logic convention''. We will discuss this very powerful designer's technique later on.

Incidentally, although Fig. 1 and Fig. 2 both show three switches used to represent three logical ''input'' quantities or conditions, neither the AND nor the OR function is restricted to three inputs. Both apply for any number of inputs from two upward.

The third fundamental logic function is called the NOT function, as mentioned earlier. This is rather different from the AND and OR functions in that it labels any situation where one event or state-of-affairs is basically the logical opposite or ''complement'' of another. In other words, a situation where one thing happens whenever another does not, and vice-versa.

A simple circuit which we can use to illustrate the NOT function is shown in Fig. 3. Here when the switch is open, the transistor is able to conduct and current will flow through the lamp. But if the switch is closed, the transistor will have no forward bias and will be cut off, extinguishing the lamp.

The relationship between the lamp being lit and the switch being closed represents the NOT function. But note that just as with the AND and OR functions, whether or not a circuit is seen to act in this way depends completely on the way we interpret its electrical behaviour in logical terms. In other words, upon how we assign the logic convention.

So that if we again consider Fig. 3, but this time look at the relationship between the lamp being extinguished and the closing of the switch, the circuit will not be seen to perform the NOT function at all. The same is true if we look at the relationship between the lamp being lit and the switch being opened.

On the other hand if we look at the fourth possible combination, the relationship between the lamp being extinguished and the switch being opened, it will again be seen to perform the NOT function. It's all a matter of interpretation, or the way we choose to assign the logic convention.

Although the logical function effectively performed by any particular electrical circuit is a matter of interpretation basic logic functions themselves—the AND, OR and NOT functions—do have quite fixed definitions. Before going any further we had better look at these definitions.

In order to do this we should first clarify what we mean by the input and output quantities of a logic element. From the logical point of view, these are not electrical quantities like current or voltage, but logical statements. These statements can have two possible values—truth and falsity. If a statement is true, it is said to have a logical value of 1; if it is false, it is said to have a logical value of 0.

Alphabetic characters can be used to represent logical statements, and this makes it possible to express logical truth or falsity in a convenient shorthand way. If a statement represented by ''A'' is true, for example, this can be written

$$A = 1$$

similarly if a statement represented by ''B'' is false, this can be written

$$B = 0$$

The logical definitions of the AND, OR and NOT functions are given in terms of input and output statements, and their truth or falsity.

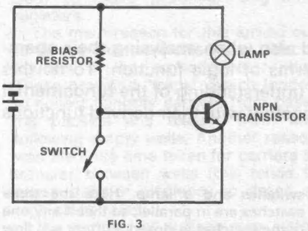


FIG. 3

Thus the AND function is defined as the function whose output statement is true (1) only when all appropriate input statements are also true, and false (0) if any or all of the input statements are false.

A very concise way of expressing this definition is by means of a ''truth table''. This is simply a table showing all of the possible combinations of truth and falsity of the input statements, and the corresponding values of the output statement. Here is the truth table definition of a three-input AND function, using A, B and C to represent the input statements, and Z to represent the output statement:

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

As you can see, it shows all of the possible combinations of truth and falsity for the three input statements, and also shows that the output statement is true only for one combination: the one where all inputs are true. For all other combinations the output is false.

Like the AND function, the OR function is also defined logically in terms of the truth and falsity of input and output statements. In this case, the OR function is defined as the function whose output statement is true (1) if any one, or more, of the input statements is true, and is false only if all of the inputs are false.

Here is the truth table definition for a three-input OR function, using the labels A, B, C and Z as before:

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note that here, the output statement Z is false only when all three of the input statements are false. It is true for all other combinations.

The logical definition of the NOT function again involves the truth and falsity of its input and output statements. The NOT function is defined as the function whose output statement is true (1) when its input statement is false (0), and vice-versa. Here is the definition in terms of a truth table, using A to represent the input statement, and Z to represent the output statement:

A	Z
0	1
1	0

Logical relationships like those involved in the three basic functions we have just defined can also be conveniently expressed symbolically, in the form of expressions and equations which look rather like those of conventional algebra. In fact this is the basis of Boolean algebra, as we will see later.

Thus the relationship between the output and inputs of a three-input AND function may be expressed by the equation:

$$Z = A.B.C.$$

where the dots represent the AND function itself, so that in words this equation says ''Z is logically equivalent to A-and-B-and-C''.

Similarly the relationship between the output and inputs of a four-input OR function may be expressed as:

$$Z = A + B + C + D$$

where the plus signs represent the OR function itself, so that in words this equation says ''Z is logically equivalent to A-or-B-or-C-or-D''.

The relationship between output and input of the NOT function may be expressed as:

$$Z = \overline{A}$$

where the ''bar'' over the A signifies the logical opposite of A, or its complement, so that in words the equation says ''Z is logically equivalent to the complement of A''.

These, then, are the three fundamental

logical functions, from which it is possible to synthesise any more complex logic function.

For convenience, however, some of the simple combinations of these fundamental functions are often also regarded as "basic" logic functions, and used in practical logic circuit modules.

One of these derived functions is the NOR function, which was originally called the "Pierce arrow function" after its originator, C. S. Pierce. This is simply a combination of the OR function and the NOT function—or if you like, it is an OR function which has the complement of its normal output.

The truth table definition of a three-input NOR function is:

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

As you can see, the output is true only when all inputs are false, and is false for all other combinations.

The relationship between output and inputs of a three-input NOR function may be expressed as:

$$Z = \overline{A + B + C}$$

which says in words "Z is logically equivalent to the complement of (A or B or C)".

A second derived logic function is the NAND function, which was originally called the "Sheffer stroke function" after its originator, H. M. Sheffer. This is similarly a combination of the AND and NOT functions—or an AND function whose output is the complement of a normal AND gate output.

The truth table definition of a three-input NAND function is:

A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Here you can see that the output is false only when all inputs are true, and is true for all other input combinations.

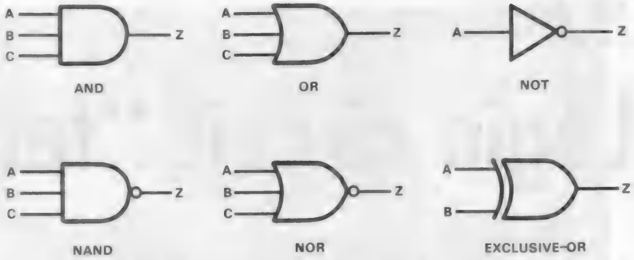


FIG. 4 BASIC LOGIC FUNCTION SYMBOLS

The output-input relationship of a three-input NAND function may be expressed as:

$$Z = \overline{A.B.C}$$

which says in words that "Z is logically equivalent to the complement of (A and B and C)".

The NOR and NAND functions are very often used as the basis for practical "building block" logic circuit modules, because it happens that either one can be used to synthesise virtually any complex logic function. This is because they both incorporate two of the fundamental functions—and as we noted earlier, AND and OR are really the opposite sides of the same logical "coin".

A third derived logic function you may come across later in digital electronics is the exclusive-OR or "ex-OR" function. This is a function whose output is true if one AND ONLY ONE of its inputs is true, but false if either none of the inputs is true, or more than one are true.

The truth table definition of a two-input exclusive-OR function is:

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

As you can see, the output is true only when a single input is true, and false for the other combinations.

The input-output relationship of a two-input exclusive-OR function may be expressed as:

$$Z = A \oplus B$$

where the plus sign enclosed by the small circle represents the exclusive-OR function itself. In words, this equation says "Z is logically equivalent to (A or B, but not both)".

The six logic functions we have looked at in this chapter are normally regarded as the basic components from which digital logic circuits are assembled. Circuit modules which perform the AND, OR, NOR, NAND or exclusive-OR functions are usually called "gates", while those which perform the NOT function ,are known as "inverters".

Incidentally it is possible to combine an exclusive-OR function with a NOT function, to produce an "exclusive-NOR" function. As you might expect, the output of this function is false if one and only one of the inputs is true, but is true if either none of the inputs is true, or more than one are true.

You might care to try working out the truth table definition of such an exclusive-NOR function, as an exercise, and compare it with that given for the exclusive-OR function.

When designing or analysing the operation of a digital logic circuit or system, it is convenient to think at least initially in terms of the logic functions performed rather than the actual circuitry. Instead of a circuit diagram, one draws or refers to a logic diagram, which has symbols showing the logic functions being performed.

The logic symbols most often used to represent the six basic logic functions we have looked at in this chapter are shown in Fig. 4. Note that these symbols strictly represent logic functions, not particular circuit modules. However in practice they are often used to represent particular modules or gates, by assuming that a certain logic convention has been assigned. We will look at this in more detail in the next chapter.

A final point. The basic logic functions we have looked at in this chapter form the basis of what is known as "combinational" logic. Circuits using combinational logic alone achieve a desired result instantaneously, using only a combination of these basic functions.

While combinational logic may be considered the backbone of digital circuits, there are many situations where the desired result cannot be achieved easily using combinational logic alone, or where the desired result itself is not simply a single event, but a series of events occurring one after the other. In such circuits it becomes necessary to supplement the basic logic functions with other elements, whose behaviour involves time.

Examples of such elements are flip-flops and monostables, which we will meet in later chapters.

Digital circuits involving these elements along with the basic logic functions are said to perform "sequential" logic as well as combinational logic.

Chapter 3

Logic circuit "families"

A number of different technologies have been evolved for producing circuit modules, and "families" of circuit modules, capable of performing often required logic functions. This chapter looks at the common logic families, and explains how they were developed as well as how they work.

In this chapter we are going to digress for a while to look at some of the types of practical circuitry which are commonly used nowadays to perform logic functions. Hopefully this will let you put basic digital theory and practice into perspective, before we go further into the subject.

As we saw in the preceding chapter, the basic logic functions can be performed simply using mechanical switches. The very earliest digital logic circuits in fact used this approach, dubbed for convenience "switch logic".

While adequate for simple circuits and systems, switch logic proved to have quite serious drawbacks. In particular, it called for very complex switches whenever the logic "input" from a switch was required to be effective in a number of different parts of the circuit.

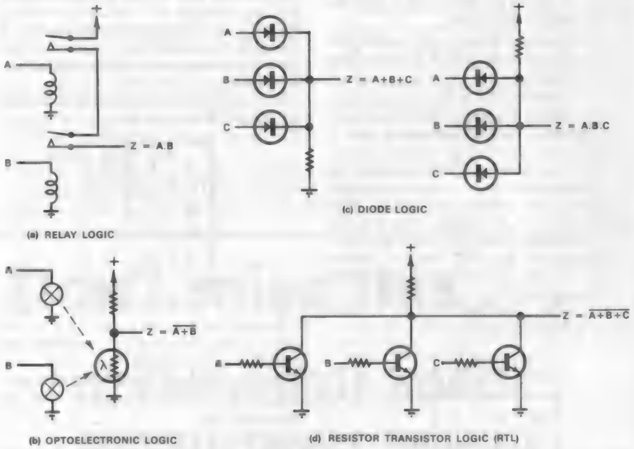
A way around this was to supplement switches with relays, which could be used to effectively multiply the number of switches poles available. Thus was born "relay logic".

While relays can provide the required number of poles, they share with switches the problem of unreliability and mechanical contact "bounce". This together with the relatively long time taken for a relay to operate has tended to restrict switch and relay logic to those applications where only a very low speed is required. Examples are the control of traffic lights and building elevators (lifts).

When the limitations of switches and relays became apparent, logic circuit designers tried other approaches. One was to use small lamps and light-sensitive resistors, giving "opto-electronic logic" (Fig. 1(b)). This avoided the unreliability and contact bounce of relays and switches, but didn't offer much else. The speed of operation was stili quite low, due to the thermal lag of the lamps and the resistive lag of the photoresistors.

The first real breakthrough came with the development of diode logic. This used semiconductor diodes, in conjunction with pull-up and pull-down resistors. Using suitable combinations of diodes and resistors, it was found possible to produce simple gates capable of performing the AND, OR and other functions (Fig. 1(c)). These had no moving parts to produce unreliability or bounce, and could thus operate at quite high speeds. The diodes introduced losses due to forward voltage drop, but these could easilv be compensated using transistor amplifiers. Simple transistor inverters could also be used to perform the NOT function where required.

Not long after diode logic came resistor-transistor logic, or "RTL" for short. This used combinations of transistors (Fig. 1(d)), with resistors as loads and input current limiters, to perform both logic and inversion.



(a) RELAY LOGIC

(b) OPTOELECTRONIC LOGIC

(c) DIODE LOGIC

(d) RESISTOR TRANSISTOR LOGIC (RTL)

FIG. 1 : EARLY TYPES OF LOGIC

RTL was the first type of logic to be used in integrated circuit logic modules, which first appeared on the commercial market at about the beginning of 1966. At first there were only three modules, but as time passed other useful modules were added to form a complete "family" of RTL devices. Each of the modules performed a useful and commonly required logic function, and all modules were designed to be electrically compatible with the other members of its family.

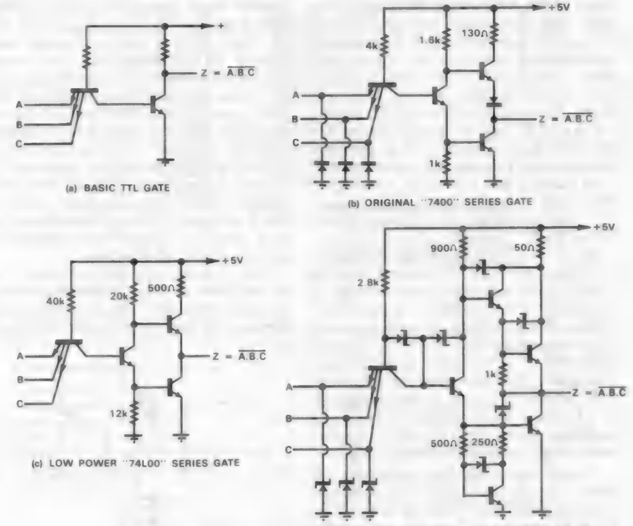
Many worthwhile logic systems were designed around RTL, which offered quite respectable performance coupled with reliability, low cost and uncritical signal and circuit requirements. However it did not lend itself to operation at speeds above about 4MHz, due to the effects of transistor charge storage and stray capacitance across the load resistors. As a result designers gradually turned away from RTL to other forms of logic as the demand grew for logic circuits capable of operating at higher speeds.

The next logic family to come into prominence was transistor-transistor logic, or TTL for short. This has been an extremely popular family of logic modules, and grew from a relatively small number of "basic gate" devices to encompass hundreds of different devices, including medium-scale integration or "MSI" types containing functional subsystems and large-scale integration or "LSI" types containing complete systems.

The basic TTL gate is shown in Fig. 2(a). It is virtually a further development of the second diode logic gate shown in Fig.

1(c), with a multi-emitter transistor replacing the diodes. A second transistor is used as an output buffer and inverter. If all of the input emitters are held at the "high" voltage state, the current flowing into the input transistor's base flows out the collector, and biases the output transistor into conduction so that its collector goes low. But if any of the input emitters is taken low, this provides a "sink" for the base current, and the output transistor is turned off. Its collector voltage thus rises to the "high" state.

If we choose to regard the "high" voltage state as logic true (1), and the "low" state as logic false (0), the gate thus performs the NAND logic function, with the output false only when all inputs are true.



(a) BASIC TTL GATE

(b) ORIGINAL "7400" SERIES GATE

(c) LOW POWER "74L00" SERIES GATE

(d) "SCHOTTKY" HIGH SPEED "74S00" SERIES GATE

FIG. 2 : VARIETIES OF TRANSISTOR-TRANSISTOR LOGIC (TTL)

Practical TTL gates are a little more complicated than this, with additional transistors and other components to increase operating speed and give improved reliability. The first practical TTL devices to appear were the well-known "54/74" series, originated by Texas Instruments. The basic circuit for a 7400 series gate is shown in Fig. 2(b). If you compare it with the basic gate, you will see that the main difference is the addition of a pair of "totem-pole" output transistors to increase the operating speed and the output current capability. Clamp diodes are also added to the inputs, to prevent damage to the multi-emitter transistor from negative-going transients.

A great many 7400-series devices based on this circuit have been produced, and used in vast numbers of highly successful digital systems. However as time progressed, circuit designers began to demand devices with either lower current consumption, faster speed, or both.

In an effort to satisfy the demands for lower current consumption the IC makers produced a family of devices known as "low-power TTL", or LPTTL, with the series designation 74L00. These were based on the same circuit configuration as standard TTL, but with the resistor values increased to reduce the current levels as shown in Fig. 2(c). This also reduced speed, but in many low power applications this was not important.

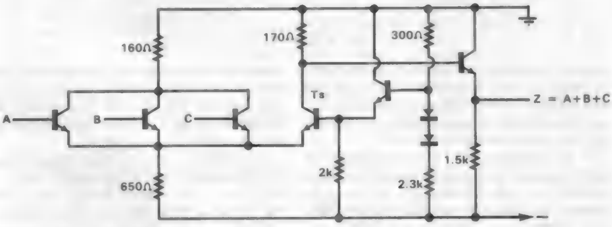


FIG. 3 : EMITTER-COUPLED LOGIC (ECL)

In order to satisfy those who demanded higher speed, they produced a "high-speed TTL" series, designated 74H00, and also a "Schottky TTL" series, designated 74S00. Like the low-power series these were still based on the original TTL circuit, but here the various resistor values were made smaller to increase speed—at the expense of higher current. With 74S00 devices the transistors were also shunted with metal-semiconductor or "Schottky barrier" diodes as shown in Fig. 2(d), to prevent them from saturating. This obviated charge storage effects, and gave a further marked increase in speed.

The most recent variant of TTL to evolve is a series of devices known as "low-power Schottky TTL", and designated 74LS00. These combine the higher resistor values of low-power TTL with the speed improvement of Schottky barrier clamp diodes across the transistors. As you might guess, they have been evolved to satisfy the demands of those wanting high speed as well as low power dissipation. The 74LS00 series offer roughly the same speed as the original 7400 series, but with the reduced dissipation of the 74L00 series. If not eclipsed by other technologies, this series seems likely to become the "standard" TTL family of the future.

Until the IC makers developed Schottky-clamped TTL, engineers designing very high speed digital systems often found that 74H00 devices were just not fast enough. As a result of their demands for extremely high speed gates and other logic elements, the makers came up with an alternative form of logic called "emitter-coupled logic" or ECL. This used a technique known as current steering, with transistors deliberately kept out of saturation to avoid the slowing-down effects of saturation.

The circuit of a basic ECL gate is shown in Fig. 3. At its heart are a number of input transistors with their collectors and emitters connected together, and with their common emitter line also connected to the emitter of a further transistor Ts. The base of this transistor is tied to a reference voltage, supplied by a temperature-compensated voltage divider and an emitter follower transistor, and its collector is connected to another emitter follower used as an output buffer.

The gate works in the following way. Due to the fixed bias voltage at its base, transistor Ts tends to conduct. This has two effects, one being that the collector current causes a voltage drop across its collector load resistor, and also at the output of the gate. The output of the gate thus goes to its "low" logic level. The second effect of Ts conducting is that it tends to act like an emitter follower, passing sufficient current into the 650ohm emitter resistor to produce a voltage drop about 0.7V less than the base voltage.

This tends to reverse bias the multiple input transistors, so that if the input bases are held at the low logic level, these transistors are cut off. However if any one or more of the input bases are taken to the high logic level, which exceeds the fixed bias on the base of Ts, then the transistor or transistors concerned are able to conduct. This passes current into the common emitter resistor, tending to raise it above the voltage produced by the emitter current of Ts, and so reverse biasing Ts. Accordingly Ts cuts off, and its collector voltage rises. The gate output voltage also rises to the "high" logic level. If we regard the "high" voltage level as logic "true" (1), this particular gate obviously performs the OR function.

As you can see, the gate effectively works by "steering" the current from the 650ohm emitter resistor either through Ts or through one or more of the input transistors. The operation does not depend upon any of the transistors being saturated, and in fact the gate is deliberately designed so that saturation cannot occur. This allows very high speed operation, and ECL devices are capable of operating at speeds greater than virtually any other technology: beyond 500MHz.

Circuit designers had pressures on them to reduce the power consumption of medium speed logic systems long before the development of low-power Schottky TTL, and as a result many turned away from TTL in an effort to find a technology capable of offering reasonable speed coupled with low power dissipation.

The technology they turned to was MOS technology, based on metal-oxide-semiconductor field effect transistors (FETs) rather than bipolar transistors. There are a number of types of logic circuit based on MOS technology, the three most widely used being NMOS, which uses predominantly N-channel MOSFETs; PMOS, which uses P-channel MOSFETs, and complementary or CMOS, which uses complementary pairs of both.

All three are currently in use at the time of writing, although NMOS and PMOS are used mainly in LSI devices incorporating complete digital systems. The technology mainly used for general-purpose small scale integration (SSI) logic modules is CMOS, which many people have described as the most elegant type of logic circuit to appear to date. It certainly offers many attractive features, including medium speed, very low current drain and low cost.

A basic CMOS gate is shown in Fig. 4(a). This is a two-input gate, and as you can see it uses only four transistors: two P-channel types in parallel at the top, and two N-channel types in series below. There are no resistors or other circuit elements.

The inputs of the transistors are tied together in complementary pairs, and shown, so that with either logic level applied to the inputs, only one transistor of each complementary pair can be conducting at any one time. If an input goes high, its N-channel transistor turns on while its P-channel transistor cuts off, and vice-versa.

If both inputs are high, both of the N-channel lower transistors conduct, while both P-channel upper transistors are cut off. The output of the gate is thus held down at the low voltage level. But if either input is taken low, this cuts off its associated N-channel transistor, breaking the path between output and low voltage. At the same time the associated P-channel transistor conducts, closing the path to the high voltage rail, and taking the output to the high logic level. This condition also applies if both inputs are taken low, so that if we regard the high voltage level as logic "true" (1), the gate may be seen to perform the NAND function.

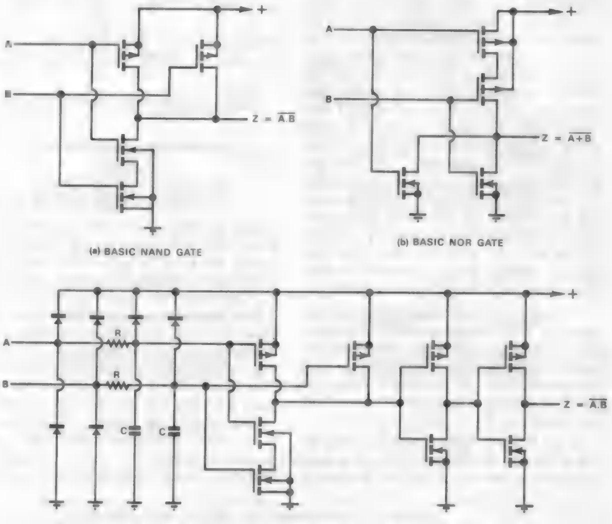
Note that the gate draws virtually no current from the supply in either of its static states, since no conducting path exists between the two supply rails. Whenever one of the upper transistors is conducting, at least one of the lower transistors is cut off. So that a CMOS gate has almost zero quiescent current drain, being an almost perfect static switching circuit.

The only time the two transistors of each complementary pair conduct together is very briefly, when changing states. This is due to differing control voltage thresholds, so that the transistor going into conduction "comes on" faster than the other transistor turns off. This means that each time the gate changes state, it momentarily becomes a low impedance path across the supply, and draws a short pulse or "spike" of current.

Because of this the average current drain and power dissipation of a CMOS gate under dynamic conditions rise from their very low quiescent levels, in a linear fashion. The power dissipation is still very modest compared with a TTL gate, however, being only about 800uW at 1MHz and 5V supply compared with about 10mW.

A basic CMOS gate which performs the NOR function if the high voltage level is regarded as logic true (1) is shown in Fig. 2(b), for you to compare with the gate just described. As you might expect from logic, it is basically a mirror image of the first gate, with the series and parallel pairing reversed.

While they perform the required logic functions, the basic CMOS gates shown in Figs. 4(a) and (b) have two main practical disadvantages. One is that the switching characteristic tends to be somewhat poor, giving indecisive switching when input logic levels do not differ by almost the full supply voltage. The other disadvantage is that without any form of input protection, the gate insulation of the transistors can easily be damaged by static charges and transients, like any other MOS devices.



(a) BASIC NAND GATE

(b) BASIC NOR GATE

(c) PRACTICAL NAND GATE WITH INPUT PROTECTION, DOUBLE BUFFERED OUTPUT

FIG. 4 : CMOS LOGIC

To get around these problems, practical CMOS devices tend to have additional circuitry like that shown in Fig. 4(c). Here the output is provided with two cascaded complementary MOS inverter stages, which give a very much sharper switching characteristic. This is known as "double buffering". The inputs are also protected from damage by means of a series R / parallel C transient filter, together with diodes which ensure that the transistor gates cannot be taken more positive than the positive rail nor more negative than the negative rail.

One of the important practical advantages of CMOS, apart from its very low power consumption, is that it can operate over a wide range in supply voltage—

typically from 5 to 15V, but some devices will operate from voltages as low as 1V.

The final type of logic to evolve to date is integrated injection logic (I^2L) or merged transistor logic (MTL), developed independently by IBM and Philips, and announced by both in early 1972. This is a type of logic which reverts back to bipolar transistor technology, but with a difference: like CMOS, there are no passive circuit elements, only transistors.

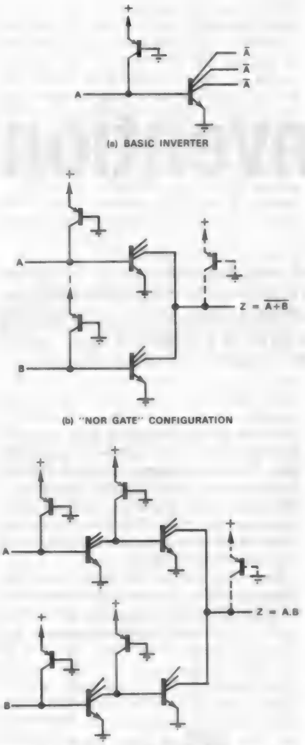
MTL was designed primarily for large-scale integration (LSI), and it is based on a multi-output inverter rather than a free-standing gate. The basic circuit of an MTL inverter is shown in Fig. 5(a).

As you can see, it has no "pull-up" resistors or transistors provided for any of the outputs, which are simply the multiple collectors of a single NPN inverter transistor. Instead, the input of each inverter is provided with a pull-up transistor, as shown. It consists of a PNP transistor connected as a current source, whose current biases its associated inverter transistor into conduction unless otherwise diverted out of the input terminal.

The reasoning behind the transfer of the pull-up element from outputs to input is that this saves the power which would otherwise be wasted in pulling up unused outputs. By having the pull-up element at all inverter inputs, bias current is only drawn where it is actually needed; unused outputs are simply left unconnected, and draw no current.

The actual logic is performed with MTL inverters by connecting together outputs from a number of inverters to the input of another, using the technique known as "wired logic" or "dot logic". This was developed with earlier types of logic, as designers found that in certain situations they could achieve logic functions such as AND and OR merely by connecting the outputs of gates together.

The idea is simpler than it may sound, as Fig. 5(b) shows. If outputs from each of the two MTL inverters are connected together, the junction will be taken down to the low voltage level if either of the inverter inputs goes high. In other words, the simple connection of the two outputs produces a potential "NOR gate", if we think in terms of the high voltage level corresponding to logic true (1). All that is needed to achieve the NOR funcuon is the pull-up action, which will be provided as soon as we connect the junction to the input of another inverter.



(a) BASIC INVERTER

(b) "NOR GATE" CONFIGURATION

(c) "AND GATE" CONFIGURATION

FIG. 5 : INTEGRATED INJECTION LOGIC (I^2L) OR MERGED-TRANSISTOR LOGIC (MTL)

Fig. 5(c) shows how a 2-input AND function may be realised using MTL elements. Here outputs from two inverters are fed to two further inverters, and outputs from each of these are again tied together. This time a "wired-AND" function is produced, however, because it will be potentially able to rise to the high voltage level only when both of the original inputs are also at high level. This corresponds to the AND function if the high voltage level is regarded as logic true (1).

Note that with this configuration, there would be nothing to stop us from using the inverted inputs available at the outputs of the first inverters, as well as the AND output. It is this "multiple function" capability of MTL which makes it particularly well suited for large-scale integration of complete logic systems.

Another feature which makes MTL ideal for LSI devices is that it involves very few IC fabrication steps — even fewer than for CMOS. This is because the PNP transistor current sources are actually "lateral" transistors, formed in the silicon IC chip simply by placing the base diffusion "island" of the NPN inverter transistor near a common P-type region "busline" connected to the positive supply. An MTL inverter is thus little more than a single NPN transistor structure, and is inherently both small and self-isolating.

A further significant feature of MTL is that its power dissipation for a given speed is even lower than CMOS. In fact MTL has been shown to have the lowest theoretical speed-power product of any logic technology yet developed, with a figure of .001 picojoules compared with about 0.1pJ for CMOS and about 100pJ for standard TTL. Like CMOS, MTL will also work at low voltages — down to about 1V.

In this chapter we have looked at the various "families" of logic devices which have evolved to date, and the technology on which they are based. I hope this helps you visualise the direct electronic implementation of the basic logic functions introduced in chapter 2, and thus gives you a solid foundation on which to build as we go further into logic system design.

Chapter 4

Logic convention & laws

To design digital circuits in general, and logic circuits in particular, it is almost essential to have a sound grasp of two important subjects. One is logic polarity convention, which determines the function actually performed by a circuit module. The other is the laws of logic, which are usually defined symbolically as the axioms of Boolean Algebra.

In chapter 2, you may recall, we noted that there is no fixed and immutable relationship between logical quantities or values and the electrical values or events used to represent them. The relationship may be chosen at will, and varied as desired, in order to achieve a desired logic function in the most efficient manner.

The relationship chosen and assigned to a particular circuit or part of a circuit is described as the "logic convention". There are two main logic conventions, one more or less the opposite of the other. These are defined in terms of the logic values "truth" (1) and "falsity" (0), and the two electrical voltage levels which are assigned to correspond to them.

If the more positive of the voltages—the "high" level—is taken to represent 1, and the more negative or "low" level to represent 0, this is described as the "positive logic convention". Not surprisingly if the opposite scheme is used, with 1 represented by the low voltage level and 0 by the high voltage level this is described as the "negative logic convention".

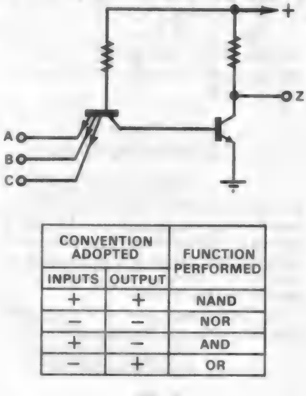
A particular logic circuit may use the positive convention throughout, the negative convention throughout, or a combination of the two. In fact the third of these possibilities is the most common, as it tends to allow the most simplification of the circuitry.

The point to grasp here is that we are entirely free to assign logic conventions as we may see fit, varying them where necessary. And as noted in chapter 2, the logic convention we use determines the logical function performed by any particular circuit module such as a gate.

In short, gates DO NOT have a fixed logic function, contrary to what you may have been led to believe by oversimplified explanations, or by the practice of manufacturers who often give gates a nominal logic function (usually the function corresponding to the positive logic convention).

This is a very important point, and deserves further emphasis. If you look at the top of Fig. 1 you will see a basic TTL gate. The electrical operation of this circuit is such that the output voltage is high (more positive) if any of the inputs is taken low (more negative). The output goes low (more negative) only when all of the inputs are high (more positive).

From this it should be fairly clear that if we adopt the positive logic convention for both inputs and output, the gate action will correspond to the NAND function. And this is the nominal logic description often given to the gate in manufacturers' literature. But it is really only one of the possibilities.



CONVENTION ADOPTED		FUNCTION PERFORMED
INPUTS	OUTPUT	
+	+	NAND
–	–	NOR
+	–	AND
–	+	OR

FIG. 1

If we choose to adopt the negative convention at both inputs and output, for example, the gate action will now correspond to the NOR function. But we can also choose to adopt the positive convention at the inputs, and the negative convention at the output—in which case the gate will effectively change into an AND gate. And last but not least, we can adopt the negative convention at the inputs, and the positive convention at the output, when it will become an OR gate.

The four possibilities are summarised in the table in Fig. 1, to help you to visualise the concepts involved.

By the way, don't confuse logic convention with power supply earthing. some logic circuits use an earthed negative supply rail, while others use an earthed positive rail. But this doesn't alter the logic convention, because voltage "high" is taken to mean the same as "more positive", and "low" to mean "more negative". It is therefore, quite feasible to use the positive logic convention in a system having an earthed positive rail, and vice-versa.

Hopefully you can see now why logic convention is important, because of its effective control over the exact logic function performed by a gate or other element. By choosing a suitable logic convention we can make a gate perform almost any desired logic function, and we can change the function a gate performs merely by assigning different logic conventions to its inputs and output.

Some readers may have already realised that this tremendous flexibility stems from the fact that logic functions like AND and OR are really only the opposite sides of the same logical "coin". If you like, AND and OR are merely different ways of looking at the same logical reality. They are in fact capable of being substituted for one another in some situations, providing one follows certain well-defined logical rules or "laws".

It is very worthwhile having a basic knowledge of these laws of logic, as they can help considerably when you are trying to simplify and rationalise logic circuits. They are usually defined symbolically, as algebraic expressions, and collectively they are known as the axioms of Boolean Algebra—in honour of George Boole.

Actually the law relating the AND and OR functions is known as the "law of duality" or "deMorgan's theorem", in honour of Boole's contemporary Augustus deMorgan. It is defined symbolically by the two expressions:

$$(\overline{A + B}) = \overline{A} . \overline{B} \qquad \ldots(1)$$

$$\text{and } (\overline{A . B}) = \overline{A} + \overline{B} \qquad \ldots(2)$$

In words, these define the theorem in terms of two complementary aspects. Expression (1) says that the logical complement of an OR function between two or more terms is equivalent to an AND function between the individual complements of the original terms. Expression (2) says that the dual of this is also true, that the logical complement of an AND function between two or more terms is equivalent to an OR function between the individual complements of the original terms.

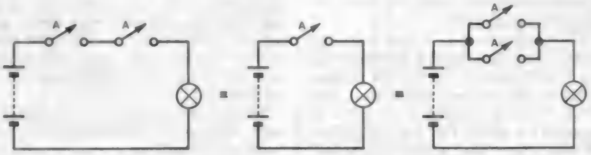
The axiomatic nature of deMorgan's theorem can be demonstrated using a truth table. The aspect expressed by expression (1) can be demonstrated by the following table:

ORIGINAL TERMS		COMPLEMENTS		COMPOSITE FUNCTIONS		
A	B	$\overline{A}$	$\overline{B}$	A+B	$\overline{(A+B)}$	$\overline{A}.\overline{B}$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

As you can see, the last two columns have identical truth values in all rows, showing that the two functions concerned are in the logical sense identical.

A similar truth table can be drawn to demonstrate the axiomatic nature of expression (2). You may care to do this, as the exercise will help reinforce your understanding of this very important theorem.

Incidentally, did you notice the technique we just used to see whether two logical functions were identical? We worked out their composite truth values for all truth value combinations of their terms, and compared them. If both functions have the same truth values for all combinations of the truth-values of their terms, then from a logical point of view they are identical and we are entitled to substitute one for the other whenever this is convenient.



A.A = A = A+A

FIG. 2

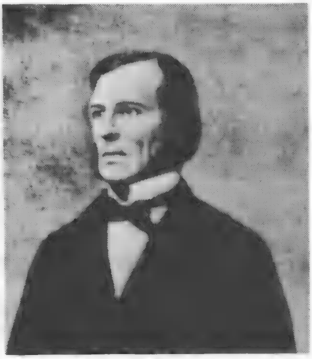
Another two important laws of Boolean Algebra are known as the "laws of expansion". These are defined symbolically by the following two expressions:

$$(A \cdot B) + (A \cdot \overline{B}) = A \qquad ....(3)$$

$$(A + B) \cdot (A + \overline{B}) = A \qquad ....(4)$$

What expression (3) says is that if we have an OR function involving two composite AND terms, and the two AND terms are identical apart from one having a term B, and the other its complement B-bar, then the term B is really redundant—so that the overall OR function is equivalent to the remaining common terms of the two AND functions.

Similarly expression (4) says that if we have an AND function involving two composite OR terms, and the OR terms are identical apart from one having a term B, and the other its complement B-bar, then the term B is again redundant and the overall AND function may be reduced to the remaining common term(s) of the two OR functions.



George Boole, the 19th century mathematician whose "laws of thought" formed the basis of modern Boolean Algebra.

Again it is possible to demonstrate that these are in fact axiomatic laws, by means of a truth table. Here is the truth table for the law of expression (3):

A	B	$\overline{B}$	A.B	A.$\overline{B}$	(A.B)+(A.$\overline{B}$)
0	0	1	0	0	0
0	1	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1

Here you can see that the first and last columns have identical truth values for all combinations, so that they are in fact logically identical. I suggest that you try drawing up a similar table to demonstrate for yourself that expression (4) is also axiomatic.

Two further laws of Boolean Algebra are known as the "laws of tautology". These are defined by the following expressions:

$$A + A = A \qquad ...(5)$$
$$A . A = A \qquad ...(6)$$

Here expression (5) simply says that if we have an OR function involving two identical terms, we are simply stuttering, in that the function is logically equivalent to the duplicated term. Similarly expression (6) says that the same thing applies if we have an AND function involving two identical terms. In both cases we are entitled to replace the composite function by the duplicated term, as illustrated by the simple diagrams of Fig. 2.

You may care to verify these laws as before using truth tables, although they are fairly self-evident. This is also true with the remaining laws of Boolean Algebra, which we will now list briefly in their symbolic form:

The "laws of commutation" point out that the order of terms makes no difference in either the AND or OR functions:

$$A + B = B + A \qquad ...(7)$$
$$A . B = B . A \qquad ...(8)$$

The "laws of association" show that the order of carrying out successive functions of the same type is similarly unimportant:

$$(A + B) + C = A + (B + C) \qquad ...(9)$$
$$(A . B) . C = A . (B . C) \qquad ...(10)$$

The "laws of distribution" show that the AND and OR functions are distributive over each other:

$$A + (B . C) = (A + B) . (A + C) \qquad ...(11)$$
$$A . (B + C) = (A . B) + (A . C) \qquad ...(12)$$

The "laws of absorption" show that both OR and AND functions which contain a common term are equivalent to that common term:

$$A + (A . B) = A \qquad ...(13)$$
$$A . (A + B) = A \qquad ...(14)$$

The "law of double negation" is simply a formal statement of the truism that a term is the complement of its complement, so that a double logical inversion restores the original term:

$$\text{not } (\overline{A}) = A \qquad ...(15)$$

The definitions of the three final pairs of laws employ the symbols 1 and 0 to represent not just truth and falsity, but terms which are always true or always false respectively.

The "laws of the universe class" show that if one term of an OR function is always true, then the OR function itself is always true. Similarly if an AND function has one term which is always true, then the AND function is logically equivalent to the remaining term(s):

$$A + 1 = 1 \qquad ...(16)$$
$$A . 1 = A \qquad ...(17)$$

This is illustrated by the simple diagrams of Fig. 3.

Conversely the "laws of the null class" show that an OR function in which one term is always false is equivalent to the remaining term(s), while an AND function in which one term is always false is itself always false:

$$A + 0 = A \qquad ...(18)$$
$$A . 0 = 0 \qquad ...(19)$$

And finally the "laws of complementation" show that an OR function between a term and its complement must always be true, while an AND term between a term and its complement must always be false:

$$A + \bar{A} = 1 \qquad \ldots(20)$$
$$A \,.\, \bar{A} = 0 \qquad \ldots(21)$$

In other words, either a term is true, or its complement must be true. At least one of the two must be true, but they can't both be true.

These laws of Boolean Algebra may seem rather trite and academic, but if you intend doing much design of logic circuits it would be worthwhile becoming at least broadly familiar with them. They form the basis of techniques used to simplify circuits down to their simplest and most efficient logical form—techniques known as "minimalisation". We will look at such techniques and related matters in the next chapter.

Before we end the present discussion, however, there is one more fairly important point you should note. This is that because the logic function of circuit modules is dependent upon the logic polarity convention we assign, this makes it difficult to draw completely unambiguous logic diagrams. As a result, different people and organisations have evolved different ways of drawing them.

One approach is to represent all gates and other logic modules by a symbol which represents their nominal function according to the positive logic convention—regardless of the function they may be actually performing. This makes it easy to identify the module being used, but makes it relatively hard to follow the logic.

Another approach is to adopt the opposite scheme, changing the symbol used to represent each module according to the actual logic function it is performing. Not surprisingly this makes it easier to follow the logic, but it is harder to identify the types of module. For example many logic gates are packaged in multiples, in IC's, and it can be confusing if three electrically identical 3-input gates of an IC are represented on the logic diagram by three different symbols (because they are being used for different functions).

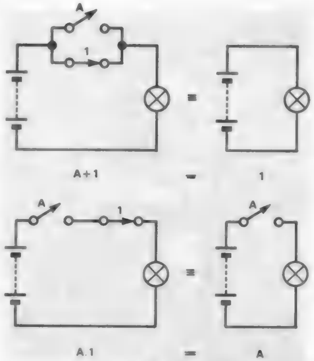


FIG. 3

Where this sort of confusion could occur, it can usually be avoided by using suitable identification labels such as "IC5a", "IC5b", etc. Providing this is done the "symbol according to function" approach is probably the more helpful of the two, and it is for this reason that we will be using the approach in this course.

However even with the "symbol by function" approach it is still difficult to represent all gate functions in a completely unambiguous manner. This is because the logic symbols in common use (those used here) do not distinguish between the internal logic functions performed by an element and the logic conventions at its terminals.

In particular the small circle or "bubble" is used by some people to indicate logical inversion or complementation within an element, and by others simply to indicate negative logic convention on input and output lines.

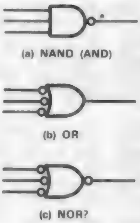
The symbols in Fig. 4 may make this clearer. The symbol in (a) is nominally that of a NAND gate, if the bubble is interpreted as part of the symbol itself. This symbol could therefore be used to represent the gate of Fig. 1 when the positive logic convention is assigned to both inputs and output.

But if we use the alternative meaning of the bubble, where it is merely an indicator that the negative logic convention applies to the line concerned, the same symbol can be used to represent the gate of Fig. 1 when it is performing the AND function.

Similarly the symbol in (b) could be used to represent the same gate when it is performing the OR function, with negative logic convention assigned to the inputs and positive logic convention to the output. Here again the bubbles at the inputs would not be seen as part of the gate itself, but as negative logic indicators on the input lines.

But we strike trouble if we try to represent the fourth application of the gate, with negative logic conventions at both inputs and output. Here it is a NOR element, which from a logical point of view involves inversion or complementation.

The trouble is, if we are already using the bubble to indicate negative logic convention at the output, we can't also use it as part of the NOR gate symbol—and vice-versa. And we can't simply use the symbol of Fig. 2(c), because this can be very misleading. It could mean that the gate performs the OR function with negative logic convention at both inputs and output, or that it performs the NOR function with positive logic at the output—and neither of these interpretations would be correct!



(a) NAND (AND)

(b) OR

(c) NOR?

FIG. 4

To resolve this sort of confusion it is really necessary to decide whether you are going to use bubbles to represent logical inversion, or negative logic convention—and then stick with your choice, at the same time stating your bubble interpretation on logic diagrams.

Chapter 5

Logic design: theory

When it is necessary to design a circuit to perform a required logic function, part of the exercise involves simplifying the logic function itself down into its most basic form—a process known as "minimalisation". This chapter gives you a broad idea of the techniques generally used to perform minimalisation.

Let us now look at what is involved in designing practical circuits to perform required logic functions. This is often called "realisation" of the logic, or "translation into hardware".

As you might expect, the aim in designing practical logic circuitry is to arrive at circuits which perform the required logic function with the least expenditure—not just in terms of initial hardware costs, but also in terms of running costs. Broadly speaking we want to produce hardware which may be manufactured as cheaply as possible, yet which will work as reliably as possible and use the least possible amount of energy.

The initial approach of logic circuit designers was to assume that this goal would be automatically achieved if the required logic function were first to be simplified down into its most basic logical form, and then merely implemented by suitable logic modules. Traditional textbooks on digital electronics have therefore treated logic design as if it consisted almost entirely of techniques to simplify or "minimalise" the required logic functions.

Unfortunately while this approach may have been reasonably valid in the early days of logic design, it has become less valid with each advance in technology since then. Soon after the first integrated circuit logic modules were developed, it became clear that these devices imposed practical constraints which could often make it difficult, clumsy and unattractive to implement a theoretically "minimal" logic function.

In short, it became necessary to temper the "ivory tower" theoretical approach with practical considerations such as module availability and cost, and the economic advantages of having a minimum parts inventory rather than merely "minimum gates count".

More recent developments such as the evolution of programmable logic arrays (PLA's) and microprocessors (uP's) have tended to reduce the significance of logic function minimalisation still further, by making it increasingly less necessary (and/or less attractive) to design custom logic circuits at all. We will look at these developments in more detail later on.

The implication of these developments is that nowadays it is neither necessary nor desirable to place heavy emphasis on logic minimalisation, in a course of this type. We therefore propose to look at this subject only in a broad and general way, sufficient to give you enough knowledge of the principles involved to allow further study if this ever proves necessary. This treatment will be followed in the next chapter by a discussion of the more practical aspects of logic design, including a look at the implications of PLA's and uP's.

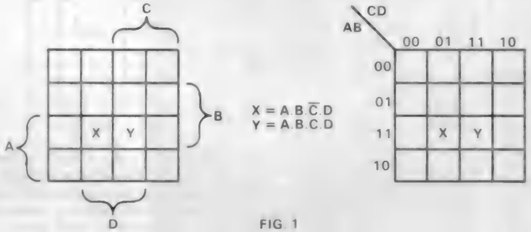


FIG. 1

To begin, then: Logically the idea behind minimalisation is to boil the required complex logic function down into its barest essentials—the minimum number of logic terms still sufficient to perform the required job. These logic terms are sometimes called the "prime implicants" of the function.

Not surprisingly, this can be achieved in a number of ways. Perhaps the most obvious way is to use the time-honoured "cut and try" approach with the logic diagram of the function, re-arranging and modifying it until no further simplification can be envisaged. While superficially attractive, this approach tends to be rather inefficient where complex functions are involved, and it is difficult to recognise when one has really arrived at the simplest form.

A somewhat more scientific approach is to manipulate the logic expression of the required function, using the various axioms of Boolean algebra. This can not only be more efficient, but the concise nature of the resulting symbolic expression can make it somewhat easier to verify that you have indeed found the most basic form.

Unfortunately when there are more than about four input terms or variables, the expression for the required logic function can become quite unwieldy, and hard to manipulate. Because of this, various graphical and topological methods have been evolved. Undoubtedly the most popular of these is the Karnaugh map method, developed by G. Karnaugh in 1953 from earlier topological map techniques used by Veitch, Venn and Euler.

A Karnaugh map is basically a multi-compartment grid or matrix diagram, in which each compartment or "cell" is made to correspond to a specific possible truth value combination of the input terms concerned. There is a cell for every possible truth value combination of the terms, so that a map for functions involving 4 terms has 16 cells and so on.

A Karnaugh map for 4 terms is shown in Fig. 1, which also shows two alternative ways of labelling the rows and columns in terms of the truth values of the terms. One way uses brackets to indicate the rows or columns corresponding to the truth of each term, while the other way labels each row or column with combinations of 0's and 1's to indicate the same thing.

Note that the two adjacent cells X and Y have truth value combinations which differ only in respect to the truth value of one term—term C, in this case. This relationship applies with all adjacent cells in a Karnaugh map, as a result of the way in which the truth values of the the terms are assigned to the various rows and columns, and it is this property which makes the Karnaugh map so useful in simplifying logic functions.

Fig. 2 shows basic Karnaugh map formats for functions involving 2 and 5 functions. The 2-term map is shown purely for illustration, because a function involving only two terms could be minimalised quite easily without a Karnaugh map. Note that the 5-term map is actually drawn as two grids, one corresponding to the truth of the fifth term "E", and the other to its falsity.

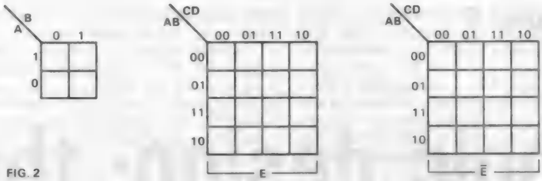


FIG. 2

To use a Karnaugh map, we mark in each cell of the grid or grids the required truth value of the output of our logic circuit, for that particular combination of truth values for the input terms. If the output should be true, we put a 1; if it should be false, we put a 0; and if we don't care (perhaps because the combination can't occur anyway), we put an "X".

When this is done, the map becomes a highly concise and easily visualised representation of the function we want to implement. It becomes relatively easy to separate the logical "wood" from the "trees", as it were, and to try various ways of defining the required result in terms of the smallest number of basic logic relationships.

This is probably best shown by an example. Let's say that we want a logic circuit whose output Z must be true for the following truth value combinations of its four input terms A,B,C and D:

$$Z = A.B.C.D + A.\overline{B}.C.D + A.B.\overline{C}.D + \overline{A}.B.C.D + \overline{A}.B.C.\overline{D} + \overline{A}.B.\overline{C}.D + \overline{A}.B.\overline{C}.\overline{D} + \overline{A.B.C}.D \quad \ldots (1)$$

Notice that we have placed OR symbols between the various combinations, to signify that any combination should be capable of producing a true output.

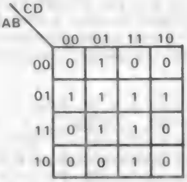
Our next step is to draw up a Karnaugh map grid for the number of terms involved —here four, and place a 1 in each of the cells which correspond to the truth value combinations shown in (1). To keep the example simple, let us assume that the output of the circuit should be false for all remaining combinations, so that we can place a 0 in all of the other cells. We thus end up with a map of the required function, as shown in Fig. 3.

Having produced the map, we are now in a position to try finding the most concise way of describing the required function. This is done by attempting to group all of the cells containing a 1 into a minimum number of sub-groups, with each of these having as many terms redundant as possible.

You may perhaps recall from the preceding chapter that a term is logically redundant in a function or expression if it may be either true or false without affecting the overall truth value.

The way this grouping is tackled on the Karnaugh map is to first try isolating any cells which cannot be grouped with any others. Then "loops of two cells" are made wherever there are pairs of adjacent cells which will not obviously form part of larger groups. Then "loops of four cells" are attempted, if there are any four-cell groups which do not seem likely to lend themselves to amalgamation into larger groups. And so on.

Note that it is only possible to group cells in loops enclosing a number of cells corresponding to an integral power of 2—2, 4, 8, 16, etc. This is because what we are doing in grouping the cells is based on the idea of making one or more of the variables redundant for each group. That term or terms must therefore be represented in the group by an equal number of "true" and "false" cells.

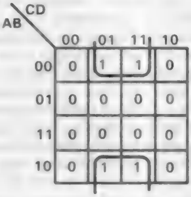


AB \ CD	00	01	11	10
00	0	1	0	0
01	1	1	1	1
11	0	1	1	0
10	0	0	1	0

FIG. 3

In doing our grouping, we can have intersecting loops—in other words, a cell can be included in more than one group. Strictly speaking this is inefficient, but at the same time it may allow us to cover the overall function with a smaller number of groups than otherwise, or the groups may be simpler in that they may involve fewer terms.

If we happen to have cells on the grid containing "don't care" X's, these can be included in loops or left out as convenient. Including them may enable us to form the cells which do contain 1's into fewer but larger loops than otherwise, which is desirable. On other occasions, regarding the cells with X's as if they contained 0's may again allow us to group the 1 cells into a smaller number of larger loops. In other words, cells with X's may generally be treated as if they contained either a 1 or a 0, whichever helps in forming the smallest number of loops enclosing as many cells as possible.



AB \ CD	00	01	11	10
00	0	1	1	0
01	0	0	0	0
11	0	0	0	0
10	0	1	1	0

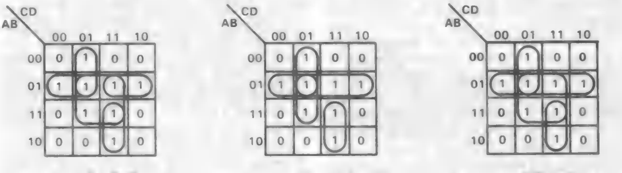
FIG. 4

It is also quite legitimate to make loops which group cells on opposite sides of the map, as shown in Fig. 4. This is because strictly speaking a Karnaugh map has no "sides", being a topological surface equivalent to the surface of a sphere. The cells which appear to be on opposite sides are thus really adjacent to each other, so that if required they may be grouped together by a single loop. The loop shown in the example of Fig. 4 corresponds to the function:

$$Z = \overline{B}.D \quad \ldots (2)$$

This is because the loop has two cells where term A is true, and two cells where it is false, making term A redundant. The same applies for term C, which is therefore also redundant, leaving only terms B and D.

As it happens, our example of Fig. 3 does not involve this type of loop. However it does have a number of possible "correct answers", which is a situation that often occurs. In other words, a complex function may not have just a single "minimal" form, but may have a number of minimalised forms which from the logical point of view may be equally basic. Which of these may be most desirable from a practical point of view will usually depend upon considerations other than



$Z = A.C.D+\overline{A}.B.\overline{C}+\overline{A}.B.C + \overline{A}.\overline{C}.D+B.D$

(a)

$Z = A.C.D+B.\overline{C}.D+\overline{A}.\overline{C}.D+\overline{A}.B$

(b)

$Z = A.C.D+\overline{A}.\overline{C}.D+\overline{A}.B+B.D$

(c)

FIG. 5

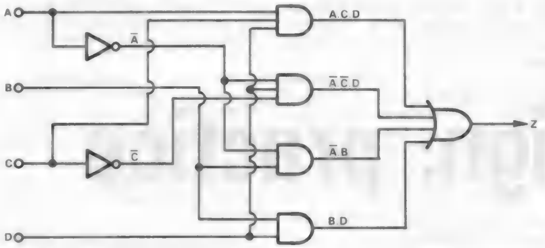


FIG. 6 $Z = A.C.D + \bar{A}.\bar{C}.D + \bar{A}.B + B.D$

sheer logic simplicity, elegance or what might be called "efficiency".

Three of the possible groupings for the example of Fig. 3 are shown in Fig. 5, together with the corresponding simplified logic expressions. Of the three probably the solution in (c) would be the most attractive from ■ practical point of view, because it results in an expression containing only four main terms, two of which contain only two of the original input terms.

By the way, note that the terms containing only two of the original input terms correspond to the loops on the map enclosing groups of four cells, while those with three original terms correspond to loops enclosing groups of two cells. This sort of relationship holds for all Karnaugh maps, in that the larger the number of cells enclosed by a loop, the smaller the number of original terms in the corresponding term of the final logic expression. So that the idea is to end up with as few loops as possible, with each loop enclosing as many cells as possible.

The direct logic diagram equivalent to the map looping and symbolic expression of Fig. 5(c) is shown in Fig. 6. As you can see, it would call for seven logic elements in all: ■ single 4-input OR gate, two 3-input and two 2-input AND gates, and two inverters.

It should be stressed, however, that this would not necessarily be the most effective or economical way of implementing the logic function we set out to turn into hardware. It is merely a logic configuration based on (hopefully) the simplest and most elegant logical expression of the job to be done. As we noted earlier, this may or may not correspond to the best answer from a practical point of view.

In the next chapter, we will look at the various practical considerations which must be taken into account in arriving at the final logic design, after one has gone through the process of logic minimalisation we have just described. We will also look briefly at the implications of recent developments in technology, which are making it less and less attractive or economic to embark upon this whole process of custom logic circuit design, at least in the traditional way.

Chapter 6

Logic design: practice

There are a number of practical considerations which can make it either unattractive or insufficient to produce a logic circuit by simply translating its minimalised logic function into hardware. In this chapter we take a look at the practical considerations, including those which arise from recent developments in IC technology.

As we noted in the previous chapter, designing a practical logic circuit is generally not just a matter of minimalising the logic function to be performed, and simply translating it into hardware. Usually, there are other considerations, of a more practical nature. At times these can make it either necessary or desirable to change the logic configuration from the minimal form, into a form which may seem superficially to be less elegant.

One of these practical considerations is that whereas logic theory treats each circuit element as a separate entity, practical logic elements usually come in packages containing multiple elements. Typical packages contain four 2-input gates (a "quad gate" package), or three 3-input gates, or six inverters (a "hex inverter" package).

Because of this, a logic configuration which is minimal from the theoretical point of view may turn out to be wasteful in practice. It may call for only a small number of elements, but these may have to be provided by quite a few different IC packages, each of which may end up with unused elements. It may therefore be better from a practical point of view to change the logic configuration, even adding elements if necessary, in order to use IC packages more efficiently.

Fig. 1 should help to make this clear. Both (a) and (b) show ways of implementing a logic circuit to produce the function

$$Z = A.\overline{B}.(C + D) + \overline{E}$$

The configuration in (a) uses only six logic elements in all, and is probably the minimal form. However it involves the use of three different IC packages, each of which is only partly used. Unless the unused elements could be used elsewhere in a logic system, this way of implementing the function would therefore be quite wasteful.

If the configuration is changed to that in (b), it becomes less minimal from the theoretical point of view. We are now using eight elements to produce the required function, instead of six. But by implementing it in this way, we require only two IC packages, instead of three, and there are only two unused elements. This could well be a worthwhile saving.

This is not to say that the goal in logic circuit design is to always end up with the absolute minimum number of IC packages, or lowest "can count", regardless of anything else. There is more to it than that, as we shall see.

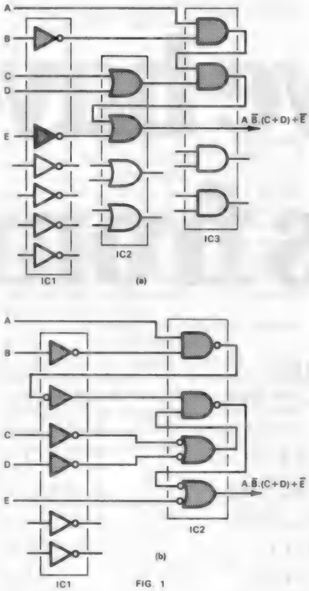


FIG. 1

In passing, note that whereas the configuration in Fig. 1 (a) uses positive logic convention throughout, the configuration (b) takes advantage of the ability to change logic convention as desired. This allows two of the elements in IC2 to be used as AND gates, while the other two are used as OR gates. Juggling logic conventions can be a very powerful tool in practical logic circuit design.

Another important consideration is component availability. Within a given family of logic ICs, there can be quite wide variations in device availability. Some device types may be readily available, while others may be in short supply. Some may be made by a number of different firms, or "multiple sourced", so that they may have a more assured availability than others which may be made by only one or two firms.

If a minimalised logic function would require the use of devices whose availability cannot be sufficiently assured, it would generally be desirable to change it to a configuration using more readily available devices. In other words, it may again be desirable to sacrifice some degree of logic elegance, in order to produce a circuit which uses readily available parts.

A closely related consideration is component inventory. As with most other parts, the price of logic element ICs tends to be inversely proportional to quantity. The more you buy, the lower the unit price. And while this can occur with "mixed orders"—i.e., orders involving a number of different parts—it tends to occur to an even greater extent with orders involving a single device type. This is not surprising, because the manufacturer's costs are lower when there is a large and uninterrupted production, testing and packaging run.

It can therefore be significantly cheaper to buy larger quantities of a relatively small number of different device types, than to buy smaller quantities of a larger number of different device types. And having bought the devices, the cost of controlling and maintaining the parts inventory may also be lower.

As a result, if the minimalised logic function would call for a relatively large number of different devices, it may well be desirable to change to a configuration using fewer different devices—even though it may have a higher "can count".

Another important practical consideration is logic element loading. Ideally, a logic gate or other element would be capable of performing its designated function regardless of the number of other elements connected to its inputs and outputs. However practical logic elements fall somewhat short of this ideal, having quite well-defined loading limitations. If these limitations are exceeded, the performance of the element generally deteriorates in one of a number of ways.

With TTL, ECL, and the earlier RTL elements, the limitation is primarily a matter of the logic levels at the output of an element tending to converge as more and more succeeding inputs are connected to it. The "low" logic level tends to rise, while the "high" level tends to fall. Naturally enough the more the two levels approach one another, the less reliable circuit operation becomes due to noise, etc. There is thus a very definite limit on the number of succeeding inputs which should be connected to an output.

In the case of CMOS logic elements, there is no direct tendency for the static output levels of an element to converge as more inputs are applied, because CMOS inputs draw virtually no static current. However they do possess capacitance, which tends to impose dynamic loading on any output to which they are connected. As capacitive loading is applied to a CMOS output, its propagation delay and dynamic power dissipation both tend to increase. So again there tends to be a limit to the number of inputs which should be connected to an output.

MTL elements have as yet only been used in LSI applications, and it seems unlikely that they will be packaged as individual logic elements, suitable for custom design of logic circuits. However if this were to occur, there would be similar loading restrictions. Here it would be a matter of the number of outputs which could be connected to an input, though, because MTL uses elements with multiple outputs and single current-source inputs. This was explained in chapter 3. Due to leakage currents, each output connected to an input would tend to increase the static loading, so that the "high" logic level would drop with the number of outputs connected.

With early logic elements such as those of the RTL family, it was common to define the loading capability of element outputs in terms of "fan-out". This was simply the number of standard element inputs which it was capable of driving reliably. A gate output with a rated fan-out of 10 would therefore be capable of driving up to 10 standard inputs, with reliable operation.

As elements having different types of input circuit came into use, it became necessary to introduce a complementary term describing the loading of an individual input—its "fan-in". This was again expressed as a multiple of the so-called standard input, so that a standard input would have a fan- in rating of 1 while an input of a low-power element might have a fan-in of 0.3. Conversely the input of a high power element might have a fan-in of 5.

Fan-out and fan-in were useful concepts, making it quite easy to work out how many inputs of various types could be connected to a particular output. All that was necessary was to add up all the input fan-in ratings, and make sure they did not exceed the output's fan-out rating. This is illustrated in Fig. 2.

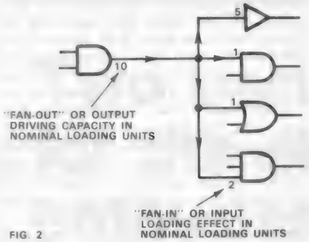

5
1
1
10
2
"FAN-OUT" OR OUTPUT DRIVING CAPACITY IN NOMINAL LOADING UNITS
"FAN-IN" OR INPUT LOADING EFFECT IN NOMINAL LOADING UNITS

FIG. 2

Unfortunately the concepts of fan-out and fan-in are less suitable for TTL and CMOS elements, for different reasons. With TTL devices, there can quite often be a significant difference between the loading capability of an element output in the "high" state and its loading capability in the "low" state. Similarly an element input can impose a loading in the "high" state which is quite different from that of another input with the same "low" state loading.

This makes it difficult to assign simple fan-out and fan-in figures to TTL devices, and manufacturers have tended to adopt an alternative approach. Outputs are rated in terms of their current sinking capability in the "low" state, and their current sourcing capability in the "high" state. Similarly inputs are rated in terms of their leakage current in the "high" state, and their operating current in the "low" state. With all of these values known, it is again possible to make sure that you are working within the loading limitations.

In the case of CMOS elements, a fixed fan-out and fan-in figure again cannot be applied because there is no static limitation. The permissible loading depends upon the speed you want to work at, and the allowable power dissipation of the devices concerned. If you are designing a low speed circuit, more inputs can be connected to a given CMOS output than if the same device is used in a high speed circuit.

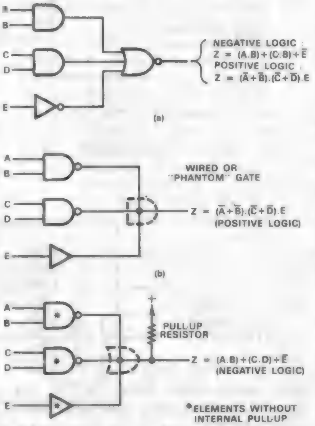

A
B
C
D
E
NEGATIVE LOGIC:
$Z = (A.B)+(C.B)+\bar{E}$
POSITIVE LOGIC:
$Z = (\bar{A}+\bar{B}).(\bar{C}+\bar{D}).E$
(a)
A
B
C
D
E
WIRED OR
"PHANTOM" GATE
$Z = (\bar{A}+\bar{B}).(\bar{C}+\bar{D}).E$
(POSITIVE LOGIC)
(b)
A
B
C
D
E
+
PULL-UP RESISTOR
$Z = (A.B)+(C.D)+\bar{E}$
(NEGATIVE LOGIC)
*ELEMENTS WITHOUT INTERNAL PULL-UP
(c)

FIG. 3

For CMOS elements, then, loading information is generally expressed in terms of graphs showing the relationship between output stage power dissipation and frequency, for various loading capacitances, and also the relationship between load capacitance and propagation delay for various supply voltages. The loading capacitance of each input is also given, together with the equivalent self-capacitance of each output, so that it again becomes possible to work out how many inputs may be connected to an output for a given operating speed.

The main point to grasp about loading is that there is a definite limit to the number of inputs which may be connected to the output of a logic element. This means that if we have a minimalised logic function which would involve exceeding the loading limits on one or more of the logic elements, it must be altered to produce a configuration which doesn't do so. Again this may well involve additional logic elements–but it is better to have a not-quite-minimal circuit which works reliably, than a minimal circuit which doesn't-quite-work under all conditions.

A further practical consideration in logic design is the opportunity to use wired or "dot" logic, in place of IC logic elements. You may recall that this type of logic was mentioned briefly in chapter 3, when we were looking at MTL.

Fairly early in the development of logic circuits, designers found that it was sometimes possible to use a direct wiring connection (or a "dot" on the schematic circuit) to perform effectively the same job as an IC logic element. In effect, a direct connection can behave as if it were a "phantom" logic element, so that the desired logic function can be performed using fewer IC elements than otherwise.

The diagrams of Fig. 3 illustrate the idea. The configuration of (a) uses a total of 4 elements, with two 2-input AND gates and an inverter feeding a 3-input NOR gate. The output represents one or other of the two functions shown, depending upon whether the NOR gate output is interpreted according to the negative or positive logic conventions. (The inputs are all assumed to follow the positive logic convention, for simplicity.)

Now compare this with the configuration of (b). This actually performs the same logic function, although it uses only three elements: two 2-input NAND gates, and a non-inverting buffer. The outputs of the three elements are joined together, and their junction effectively behaves as a "phantom" gate. If all three outputs are high, the junction is also high; but if any output goes low, it pulls the junction low also.

If we think in terms of the positive logic convention, the wired junction of the three outputs thus acts as an AND gate. Alternatively if we think in terms of negative logic, it is acting as an OR gate. Hence the terms "wired-AND" and "wired-OR", often used by designers to describe such phantom gates.

Note that although input term E no longer needs to be inverted with the configuration of (b), a buffer element is shown between the source of E and the wired junction. This is generally required to provide isolation, as a direct connection would force E to follow the junction, and prevent E from being used elsewhere. However if E is derived from another logic element, and is not required elsewhere, the buffer would not be required.

Astute readers may have already realised that if the logic elements used to drive a wired-logic junction have output stages with active pull-up devices, the operation of the wired junction involves conflict between output stages. This is because one element may have an output device attempting to keep the junction "high", while another may have another output device attempting to drive the junction "low".

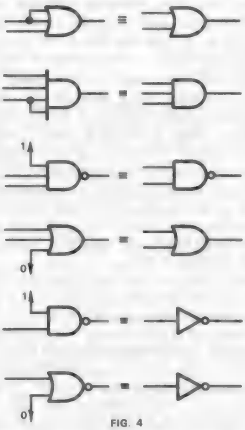


FIG. 4

While this is nominally an infringement of loading rules, in practice it causes no trouble with many types of logic element. For example TTL element output stages are designed to be able to "sink" very much more current in the "low" state than they are able to "source" current in the "high" state, so that a "low" output can easily force a number of other "high" outputs into the low state without causing the low voltage level to rise unduly. Naturally enough this causes increased power dissipation in the outputs that are forced low, but generally this is acceptable providing only one element of any particular IC is forced low at any one time.

Of course it is not always possible to arrange that all of the logic elements involved in driving a wired-logic junction are part of different ICs. If a number of such elements must be implemented using a single IC, and more than one may be forced low at the same time, this can still be done by using an IC whose elements have output stages without internal pull-up devices. These are often called "open-collector" or "uncommitted output" logic elements.

An external pull-up resistor is then used to provide logic "high" pull-up for the wired junction, as shown in Fig. 3(c). To emphasise that the wired junction may be interpreted as performing either a wired-AND or wired-OR function, this diagram shows a phantom OR symbol, and the negative logic version of the output function.

By the way, it may be worthwhile to point out that multi-input logic elements can almost always be used to perform the function of elements having fewer inputs. In other words, a 4-input gate can be used as a 3-input gate, or a 2-input gate, or even as a buffer or inverter. Usually this is done by either paralleling the redundant inputs with those that are used, or by tying them to an appropriate "true" or "false" logic rail, as shown in Fig. 4.

This trick can be very handy if one needs to implement a simple logic element, but to provide such an element directly would involve the addition of a complete new IC package. If there is a more complex element in an existing IC which would otherwise be "spare", this can be used to advantage.

Note that if a redundant input of a gate is not paralleled with a used input, it must be taken to the logic "true" level (1) in the case of an AND or NAND gate, or tó the logic "false" level (0) in the case of an OR or NOR gate. This is because we are really making use of the Boolean Algebra "universe class" and "null class" laws, as defined in expressions (16), (17), (18) and (19) of chapter 4.

If nothing else, the points we've raised so far in this chapter should have driven home the point that there is a lot more to logic design in practice than merely translating the minimalised logic function directly into hardware. In fact there are many things for the designer to bear in mind, so that the design of efficient custom logic circuits tends to involve a great deal of time and skill.

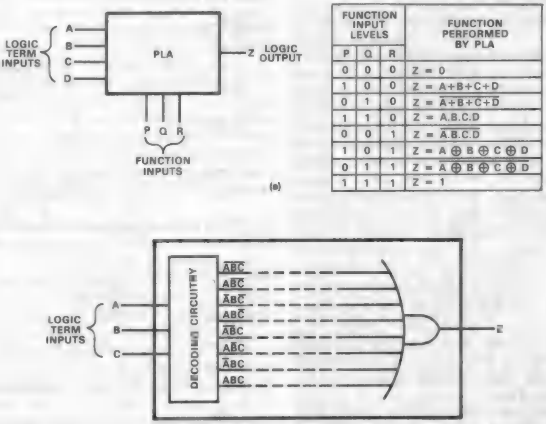
Naturally enough this tends to make custom designed logic circuits quite costly, and accordingly those involved in producing digital systems and equipment have long been keen to find an alternative approach. Until recently there was no such alternative, but the rapid advances in IC technology have now changed this position, and seem likely to change it even more in the future.

In fact two broad alternatives to custom logic circuit design are emerging. One involves the use of programmable logic arrays, or "PLA's", and the other involves the use of microprocessors or "μP's".

A PLA is basically a large-scale integrated (LSI) circuit which is capable of being programmed to perform one of a number of logic functions. At present there are two broad types, which work in rather different ways.

One type consists of a very complex logic circuit which has a "repertoire" of possible logic functions. Any one of the functions in its repertoire may be performed by applying appropriate control signals to a set of function input terminals. This is shown in Fig. 5 (a). The device may either be arranged to perform a single function in static fashion, by applying fixed control signals to the function inputs, or it may be made to perform various functions at different times, by changing the control signals as required.

This type of PLA is essentially a collection of standard logic circuits, each one of which is brought into operation as required under the control of additional logic connected to the function inputs. As such, it is most suitable for applications



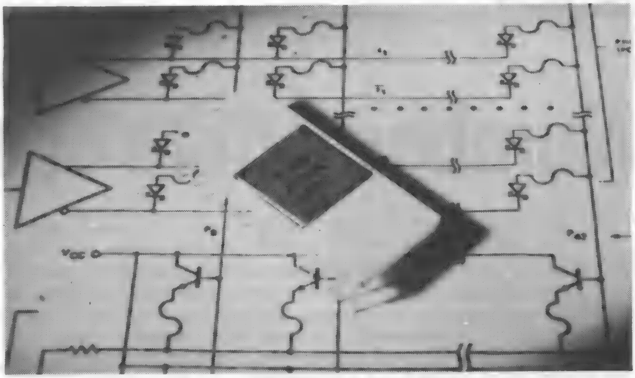
FUNCTION INPUT LEVELS			FUNCTION PERFORMED BY PLA
P	Q	R	
0	0	0	Z = 0
1	0	0	Z = A+B+C+D
0	1	0	Z = $\overline{A+B+C+D}$
1	1	0	Z = A.B.C.D
0	0	1	Z = $\overline{A.B.C.D}$
1	0	1	Z = A ⊕ B ⊕ C ⊕ D
0	1	1	Z = $\overline{A \oplus B \oplus C \oplus D}$
1	1	1	Z = 1

FIG. 5

where one mainly wants to implement one of a number of fairly standard logic functions.

Where more complex logic functions are required, the second type of PLA is more suitable. This type is basically not a logic circuit at all in the strict sense, but a "read-only memory" or "ROM". We will be looking at such devices in a later chapter, and it would not be appropriate to try and explain their operation here in any detail. However for the moment you can visualise this type of PLA as a device which is capable of separating out all of the possible truth value combinations of the input logic terms, and detecting when each combination occurs.

This allows virtually any logic function involving the input terms to be "synthesised", by arranging for the output of the PLA to be true for those truth value combinations which are appropriate, and false for the rest. The PLA is programmed internally to do this using the truth table of the required function as a guide.



This user-programmable PLA device allow the synthesis of complex logic functions involving up to 48 truth value combinations of as many as 16 input terms. It also has eight independent outputs (Signetics).

If you find this hard to understand, the diagram of Fig. 5 (b) may help. It shows a simple and hypothetical PLA of this type, involving only three input terms. This is purely to illustrate the idea, as it would be rather wasteful to use a PLA for simple functions involving only three terms.

As you can see, the PLA consists of two basic sections—"decoding circuitry" which detects each of the possible truth value combinations of the input terms (here there are 8 possible combinations), and an output circuit which functions rather like a multi-input OR gate. Each output of the decoding circuitry is potentially capable of being connected to an input of the OR gate, with the PLA being programmed by making these connections as appropriate.

For example, if we wanted the PLA to perform the same function as a normal AND gate, we would arrange for only one of the decoding circuit outputs to be internally connected to the OR gate—the one corresponding to the combination (A.B.C). The output Z would thus go true only for that combination, and remain false for the rest.

Similarly, if we wanted the PLA to perform the NOR function, we would arrange again for only one internal connection—that corresponding to all three input terms being false.

If we wanted to be a little more adventurous, we could program the PLA to perform the exclusive-OR function. This would involve linking up internally three decoder outputs, to give the effective output function:

$$Z = A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.\overline{B}.C$$

Alternatively, we could make it perform an exclusive-NOR function, by linking up the other five internal decoder outputs instead. Then the output would be false only for the three truth value combinations just given, and true for all others.

Hopefully you can see from these examples how a PLA of this type may be "programmed" to perform virtually any required logic function, simply from a knowledge of the truth table of the function required.

Most practical PLAs are designed to accept many more than three input terms. In fact typical devices are capable of handling as many as 12 or 16 input terms, detecting as many as 48 different output truth-value combinations, and producing eight separate output terms.

Complex PLAs of this type do not use the exact arrangement shown in Fig. 5(b), however, because this would involve a very elaborate internal decoder—most of which would be wasted. For example with 16 input terms, a full decoder would involve no less than 65,536 output lines! Of these only a small number would tend to be used.

To avoid this complexity, the decoder itself is made programmable. In other words, each of the input terms and their logical complements are made available via buffer stages, and the PLA is programmed to decode only those truth value combinations which are actually required to produce the required function.

Like PLA's, microprocessors or "μP's" are recent developments of large-scale integration technology. A μP is basically a miniature computer, compressed into one or more IC packages. Again we will be looking at μP's in some detail in a later chapter, and it would not be appropriate to try and explain their operation properly here. However for the sake of completeness, we should perhaps show in broad terms how a μP can be used to perform complex logic functions in place of a custom designed logic circuit.

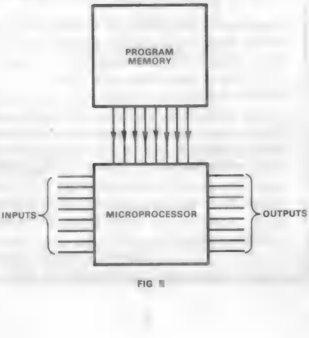


FIG. 6

A μP consists basically of a "processor", capable of performing any of a number of basic logical, arithmetic and manipulative tasks under the control of "instructions". Each instruction is a number, and by feeding the processor a series of numbers in turn, it may be made to perform a sequence of tasks. Such a series of instruction numbers is called a "program", and is usally stored in a memory device of some sort. A complete μP system thus tends to consist of the μP itself together with a memory to store the program, as shown in Fig. 6.

To use a μP to perform a complex logic function, it is programmed to behave rather like the PLA in Fig. 5.(b), only in a dynamic fashion. The sequence of instructions force it to continuously monitor the truth values of the various terms at its inputs, and compare them with a list of reference combinations stored in the memory. Then, according to the reference combination which matches the input combination each time, the μP is instructed to vary the truth value produced at one of its outputs.

The fact that the μP operates in a dynamic fashion means that there tends to be a slight delay between a change in the input terms and the appropriate change in the effective logic output. This can make the μP less attractive than the PLA where a complex logic function must be performed at very high speed. However the big advantage of the μP is that its effective logic function may be changed quite rapidly, merely by altering its stored program.

Chapter 7

Numbers, data & codes

To properly understand the operation of more complex digital elements and circuits, you need to have a good grasp of numerical notation and the codes used to handle both numeric and alphanumeric data in digital systems. In this chapter we discuss most of the codes you are likely to come across: binary, octal, hexadecimal, binary-coded-decimal, and the alphanumeric codes ASCII and Baudot.

As we saw back in chapter 1, digital circuits are capable of handling not just logic signals, but virtually any sort of information. All that is necessary to do this is to encode the information, so that it may be represented by suitable combinations of the two voltage or current levels provided in most digital circuits.

A single digital circuit can only adopt one of the two possible voltage or current states at any one time. In other words, it is only capable of conveying one tiny snippet of information at a time—such as a distinction between "hot " and "cold", "yes" or "no", "up" or "down", and so on.

This is, in fact, the smallest amount of information which can be defined. It forms the fundamental unit of information, and is given a name which you have perhaps already seen used: the "bit".

A single digital circuit element can only convey a single bit of information at any one time. But, needless to say, there are a great many practical situations where we wish to convey much more information than this. Typically quite a large number of bits may be required to convey the desired information, so that digital circuits must somehow be arranged to deal with multiple bits. In general, there are two broad ways in which this is done.

One is by sending the various bits of the information one after the other in time, through the same single digital circuit. This is known as the "serial" method. The other way is to duplicate the circuit a number of times, and use the resulting multiple circuits to convey each of the bits of information simultaneously. This is known as the "parallel" method.

Each of the two methods has its advantages. The serial method tends to save money, as it uses single circuit elements. But it also tends to be relatively slow, because of the need to shuffle the information through bit by bit. In contrast the parallel method tends to be somewhat faster, as the bits are handled at the same time, but it also tends to be more costly because of the circuit duplication.

The serial approach thus tends to be used in applications where low cost is essential, and speed is not important, while the parallel approach is favoured in applications where speed is important enough to justify the higher cost.

The two approaches are not mutually exclusive, however, and in practice a mixture of the two is generally used. This is mainly because a fully parallel approach is impractical where large amounts of information must be handled.

It is convenient to handle large amounts of information in groups of bits known as "words". The words generally consist of a fixed number of bits for any given system; typical word sizes used are of 8 bits, 12 bits, 16 bits and 32 bits. Words of 8 bits are often called "bytes".

In most digital systems the words themselves are usually handled serially, one after the other. But the bits within the words may be handled either serially or in parallel, depending upon the speed required. Thus, if a system handles information in 8-bit bytes, the bits within each byte may be dealt with either serially via a single circuit, or in parallel via 8 identical circuits: but the bytes themselves are normally dealt with serially, one after the other.

Irrespective of the way in which the digital bits and words are to be handled, some consistent method must be used to encode the actual information in digital form. This is to allow for the eventual decoding of the information, at the output end of the system, and also for ease of trouble-shooting if this is required.

In many cases, information is encoded by using each of the various possible truth-value combinations of a group of bits to represent one of the symbols normally used to convey the information. This is the encoding technique generally used for information expressed in normal language, using alphabetic and numeric (or "alphanumeric") symbols. We will look at this sort of encoding later on.

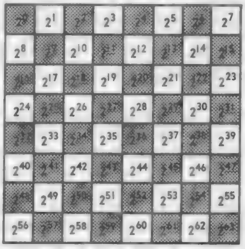
Where the information to be handled is solely numerical data, a different technique is generally used. Here the data itself is converted from its normal form, in decimal notation, into an alternative notation more suitable for manipulation by digital circuits.

The alternative notation which is very often used is **binary** notation, which uses 2 as its base or radix, in place of the 10 used in our familiar decimal system. Binary notation is very well suited for digital circuits, as it involves only two numeral values: 0 and 1. These replace the ten numeral values (0,1,2,. . . 9) used in decimal notation.

This means that many more digits must be used to represent a given number, because each numeral position represents a certain power of 2 rather than a power of 10. In place of the units-tens-hundreds-thousands progression of the decimal system, the numeral positions to the left of the "binary point" represent units, twos,

SISSA'S REWARD: binary notation can be deceptive

In antiquity, according to legend, the Grand Vizier Sissa invented the game of chess for the Indian King, Shirham. When invited by the King to name his reward, he asked only for the quantity of wheat required to cover a chess board by placing one grain on the first square, two on the second, four on the third, eight on the fourth, and so on.The King marvelled at his self-denial, until Sissa explained that the number of grains required would be only one less than the 64th power of 2: more than 18 million, million, million...



fours, eights and so on. Here are a few decimal numbers and their binary equivalents:

decimal 5 =	101
decimal 16 =	10000
decimal 50 =	110010
decimal 99 =	1100011

As you can see, the binary system is clumsier in the sense that rather more digits are required to represent the same numbers. But each numeral position only has two possible values, so that it may easily be represented by a digital bit. In fact this is where the word "bit" comes from, being contraction of the words **"binary digit"**.

Most digital computers and other digital systems handle numbers in binary form. However we human beings tend to find large binary numbers difficult to interpret, and as a result it is convenient to interpret the binary numbers in terms of one of two related notation systems.

One of these is the **octal** system, based on a radix of 8. This uses eight numerals: 0,1,2,3,4,5,6, and 7, with each digit position corresponding to a power of 8. Thus reading to the left from the "octal point", they correspond to units, eights, sixty-fours, five-hundred-and-twelves, and so on.

In itself, the octal system has no particular relevance to digital circuitry. But the fact that 8 itself corresponds to the third power of 2 means that each octal digit is equivalent to three consecutive binary digits. Or if you like, an octal digit may be used to replace each group of three, according to the following table:

octal	=	binary
0	=	000
1	=	001
2	=	010
3	=	011
4	=	100
5	=	101
6	=	110
7	=	111

A large binary number may thus be represented by its equivalent octal number, simply by starting from the right and replacing each group of three bits by its equivalent octal digit:

binary 100 011 101 010 = octal 4352
binary 111 000 110 001 = octal 7061

(in these examples the binary bits have been grouped in threes merely to make the equivalence more evident).

As you can see, the octal representation of a 12-bit binary number involves only 4 digits, making it rather easier to handle and remember as far as we humans are concerned. And the important thing is that one can convert quite easily from one notation into the other as required, after very little practice. Hence it you are presented with a number in octal notation, it is really quite easy to convert it into binary:

octal 5742 = binary 101111100010

Actually even octal arithmetic is quite easy to get used to after a little practice, being not all that different from decimal arithmetic. The main things to remember are that the numerals "8" and "9" have no significance, and that the "carry" takes place after 7:

7 + 1 = 10
77 + 1 = 100

Many computer programmers become very adept at thinking in terms of octal notation, and are able to interpret virtually all of a computer's binary operation in octal terms.

The other binary-related notation often used to represent binary numbers for the convenience of we humans is **hexadecimal**, often shortened to "hex". This is notation based on a radix of 16, and as you might expect from this, it requires 16 numeral symbols—six more than we normally have available.

Any suitable symbols could be used for the additional six numerals, but for various practical reasons it is common to use the first six letters of the alphabet: A,B,C,D,E and F. This makes hexadecimal numbers look quite strange until you get used to them, as at first sight they seem like a strange mixture of numbers and letters, or even just letters alone. For example 5C72, 63IF, C008 and FFFA are all perfectly sensible hexadecimal numbers.

Like the octal system, hexadecimal notation has no particular relevance to digital circuitry. But again, 16 corresponds to the fourth power of 2, so that each hexadecimal digit is equivalent to four consecutive binary digits. This makes hexadecimal notation very convenient for handling large binary numbers. Each group of four bits is replaced by its equivalent hexadecimal digit, thus:

hexadecimal		binary
0	=	0000
1	=	0001
2	=	0010
3	=	0011
4	=	0100
5	=	0101
6	=	0110
7	=	0111
8	=	1000
9	=	1001
A	=	1010
B	=	1011
C	=	1100
D	=	1101
E	=	1110
F	=	1111

A large binary number may thus be represented by its equivalent hexadecimal number, simply by starting from the right and replacing each group of four bits by the appropriate hexadecimal digit:

e.g. binary 1111 0110 1010 1101
= hex F6AD
binary 1110 0000 0011 1011
= hex E03B

After a while, it becomes quite easy to convert from hexadecimal to binary and vice-versa, whenever this is necessary.

The binary notation we have looked at so far in this chapter is what is often called "pure binary", where all the consecutive digit positions correspond to increasing powers of 2. This is the binary notation used to encode numerical data in a great many digital systems, but there are some applications where it is more convenient to use other binary codes. These still use only two digit values (0 and 1), but in general they weight the digit positions differently from pure binary.

One important group of these binary codes are the **binary-coded-decimal or "BCD" codes**, which tend to find use where it is either necessary or convenient to handle numerical data in binary words which directly represent the various decimal digits. Typical applications of this sort include digital instruments such as frequency counters and DVMs (digital voltmeters).

Broadly speaking, a code of this sort must use four bits to represent each decimal digit. This is because this many bits are required to provide the necessary 10 different value combinations: three bits only provide 8 different combinations. But four bits actually provide a potential of 16 different combinations, as we have seen from the table of hexadecimal equivalents.

A BCD code only uses 10 of these combinations, "wasting" the others available in each 4-bit group. This makes BCD codes less efficient than pure binary, but gives them a measure of "redundancy" —a characteristic which allows a degree of error detection. This will be discussed in a later chapter.

Perhaps the most obvious way of getting a BCD code is to use the first 10 value combinations of normal "pure" binary notation. This gives the so-called "8421 BCD" code, where the digits in the name reflect the significance or "weighting" of the four bits:

decimal	8421 BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

(the remaining 6 combinations are unused, or "illegal").

This is probaly the most often used BCD code, but it is by no means the only one possible. In fact, if you care to work out the total number of ways in which any 10 combinations out of the 16 possible could be selected to represent the 10 decimal digits, you will find that there are close to 76 million possible BCD codes!

Happily there are only a few others used apart from 8421 BCD, and even these are not encountered very often. They are mainly used in specialised situations where some special characteristic is needed, such as the need to use the least number of "1's".

Just before we leave BCD codes, it may be a good idea to give an example to show the difference between pure binary notation and 8421 BCD. It is important to realise that the two are quite different where large numbers are concerned, even though they are identical for numbers less than 10. Here is a number expressed in eight bits, with the first and second groups of four separated for convenience:

1001 0111

If this is interpreted as being in pure binary, it corresponds to the decimal number 151 or (1 + 2 + 4 + 16 + 128). But if instead we interpret it as two digits of 8421 BCD code, it corresponds to the decimal number 97.

In other words, it makes a great difference whether a number is encoded in pure binary or 8421 BCD. With pure binary, all bits have an increasing significance or weighting as a power of 2, whereas with 8421 BCD this is only true within each 4-bit group representing a decimal digit. With all BCD codes, each group of 4 bits is regarded as being quite separate from the others—even when a number of 4-bit groups are handled together as larger words, for convenience.

All of the specific binary notations and codes we have looked at so far have used fixed weighting, in which each bit carries a consistent weighting as a power of two. But this is again not essential. In fact there are situations where a binary code having variable weighting can offer decided advantages.

An example is where continuously varying analog quantities such as voltages, pressures, temperatures, shaft positions and so on must be encoded digitally, or "digitised". When this is done using a fixed-weighting code such as pure binary, there are many parts of the code where a very small change in value requires a simultaneous change in the value of many of the encoding bits. If we are using 5 bits, for example, and the value changes from decimals 15 to 16, all five bits must change in value:

15 = 01111
16 = 10000

To accurately encode such a change, the measuring sensor or circuit must be arranged so that all of the bits do in fact change values exactly in synchronism. Otherwise, the output of the encoding circuit may pass momentarily through other combinations of the bits concerned, causing errors. This can be very difficult to prevent.

The problem can be made much easier by using a variable-weighting code such as the "reflected binary" or **Gray code**, named after its originator Elisha Gray. This code is cyclic, and so arranged that every transition from one value to the next involves a change in the value of only one bit. Here are the first 16 numbers in the Gray code, with the pure binary equivalents included for comparison:

Decimal	Binary	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The reason for calling this a "reflected" code should be clear if you look at the three least significant bits, and compare the patterns of the first eight numbers with those of the last eight. As you can see, the patterns "reflect" in the second eight, being simply repeated in reverse order. The only difference is in the fourth bit position.

Normally the Gray code is used as a "pure" code, using as many bits as required to gain the necessary accuracy. Part of the code could be used as a BCD code, but it would not retain its feature of a single bit change when changing from decimal 9 back to 0. This can be overcome by using a modified Gray code, of which there are quite a number.

There are a variety of other codes used

TABLE 1: ASCII CHARACTER CODE

								0	0	0	0	1	1	1	1
								0	0	1	1	0	0	1	1
								0	1	0	1	0	1	0	1
B6	B5	B4	B3	B2	B1	B0	COLUMN → ROW ↓	0	1	2	3	4	5	6	7
			0	0	0	0	0	NUL	DLE	SPACE	0	@	P	\	p
			0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
			0	0	1	0	2	STX	DC2	”	2	B	R	b	r
			0	0	1	1	3	ETX	DC3	# (£)	3	C	S	c	s
			0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
			0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
			0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
			0	1	1	1	7	BELL	ETB	’	7	G	W	g	w
			1	0	0	0	8	BACK SPACE	CAN	(	8	H	X	h	x
			1	0	0	1	9	HOR. TAB	EM	)	9	I	Y	i	y
			1	0	1	0	10	LINE FEED	SUB	*	:	J	Z	j	z
			1	0	1	1	11	VERT. TAB	ESCAPE	+	;	K	[	k	{
			1	1	0	0	12	FORM FEED	FS	,	<	L	\	l	∫
			1	1	0	1	13	CARRIAGE RETURN	GS	—	=	M	]	m	} (ALT MODE)
			1	1	1	0	14	SHIFT OUT	RS	.	>	N	∧ (↑)	n	~
			1	1	1	1	15	SHIFT IN	US	/	?	0	_	o	DEL (RUB OUT)

for numerical data, but they are rather too specialised for us to discuss here. Many are designed to detect errors in transmission, using techniques which may involve the addition of extra bits of information to each word group.

To round off this brief look at the ways in which information is encoded in digital form, let us consider codes used to handle information expressed in alphanumeric form—i.e., in normal language. As mentioned earlier, this sort of information is encoded by using each of the various truth-value combinations of a group of bits to represent one of the symbols normally used to express the information. The encoding scheme is thus rather like that for BCD coded numeric data, but using more than 4 bits in each group because of the larger number of characters involved.

From a look at the keyboard of a typewriter, you would perhaps guess that about 40 characters would be required to convey most alphanumeric information—the 26 alphabetic characters, the 10 decimal numerals, and a few punctuation marks. In fact quite a few more than these are needed, because of the need to convey non-printing or "control" characters such as carriage return, line feed, space and so on. These are needed in order to convey information in neat and easily intelligible form.

It turns out that even for sending information in fairly crude form, something like 64 different characters are required. And if you want refinements like upper and lower case letters, brackets and a few special symbols, the list grows to around 128. This means that a code using at least six bits is required for handling basic alphanumeric information, or seven bits if full flexibility is required. (A group of 6 bits is capable of 64 different truth-value combinations, while 7 bits provide 128 different combinations.)

As with BCD codes, there are a very large number of possible codes which could be used for alphanumeric encoding. And quite a few have been used, in various applications. However we have space here to deal with only two.

The first of these is the code in most common use nowadays, known as **"ASCII"**. Pronounced "asskey", this is an acronym standing for "American Standard Code for Information Interchange". Although originated in the USA, it is now very widely used for alphanumeric encoding.

The full ASCII code uses 7 bits, to provide encoding combinations for 128 different characters. These are shown in Table 1, which is arranged in columns according to the values of the three most significant bits (B6, B5 and B4). Within each column are the 16 truth value combinations provided by the four least significant bits. Within the 128 cells formed by the 8 columns and 16 rows are the corresponding characters.

As you can see, columns 4 and 5 provide the upper case alphabetic characters, together with a few special characters. Columns 6 and 7 provide the lower case characters, while column 3 provides the numerals and some punctuation marks. Further punctuation marks are provided by column 2, together with "space" and some special symbols. Columns 0 and 1 provide non-printing or control characters, including format control characters.

TABLE ■ : BAUDOT CODE

CHARACTER		CODE				
LETTERS	FIGURES	B1	B2	B3	B4	B5
A	–	1	1	0	0	0
B	?	1	0	0	1	1
C	:	0	1	1	1	0
D	$	1	0	0	1	0
E	3	1	0	0	0	0
F	!	1	0	1	1	0
G	& (OR @)	0	1	0	1	1
H	STOP (OR £)	0	0	1	0	1
I	8	0	1	1	0	0
J	, (OR BELL)	1	1	0	1	0
K	(	1	1	1	1	0
L	)	0	1	0	0	1
M	.	0	0	1	1	1
N	,	0	0	1	1	0
O	9	0	0	0	1	1
P	0	0	1	1	0	1
Q	1	1	1	1	0	1
R	4	0	1	0	1	0
S	BELL (OR !)	1	0	1	0	0
T	5	0	0	0	0	1
U	7	1	1	1	0	0
V	; (OR =)	0	1	1	1	1
W	2	1	1	0	0	1
X	/	1	0	1	1	1
Y	6	1	0	1	0	1
Z	" (OR +)	1	0	0	0	1
SPACE		0	0	1	0	0
CARRIAGE RETURN		0	0	0	1	0
LINE FEED		0	1	0	0	0
"LETTERS"		1	1	1	1	1
"FIGURES"		1	1	0	1	1

Sometimes an abbreviated form of ASCII code is used for purposes which do not require lower case letters, etc. This uses only the bit combinations of columns 2, 3, 4 and 5, and involves only six encoding bits (bit 6 is omitted). It is often called "6-bit ASCII".

There is nothing like a little practical example to help you visualise something. Here is a very short message encoded in 7-bit ASCII, which you might like to try decoding:

```
1000111
1001111
1001111
1000100
0100000
1001101
1001111
1010010
1001110
1001001
1001110
1000111
0100001
```

Actually this particular message could have been encoded in 6-bit ASCII, as it uses only characters from columns 2, 4 and 5. And if this had been done, we could then have expressed the resultant string of 6-bit binary words as eqivalent 2-digit octal words, for convenience:

```
07
17
17
04
40
15
17
22
16
11
16
07
41
```

Have you decoded the message yet? It is simply the familiar greeting "GOOD MORNING!", complete with a space between the two words, and the final exclamation mark.

The other alphanumeric code we should look at briefly is the **Baudot code**, also called the "Murray code" and the "International Telegraphic Code No. 2". This code originates from about 1906, when teleprinter machines were developed for sending automatically encoded messages over telegraph lines. It is still in wide use throughout the world for transmission of news services, ship-shore messages, etc.

The basic encoding used in Baudot code is shown in Table 2. There are two main points to note, one being that the code follows no logical progression of the sort evident in ASCII. This makes conversion between Baudot and a progressive code like ASCII rather more complicated than it might have been.

The second point to note is that although a total of 57 characters may be transmitted, only five encoding bits are used! At first sight, this may seem to be achieving the impossible, as from theory 5 bits have only 32 truth-value combinations. But the Baudot code gets around this by using two special control characters, which work in conjunction with a "memory" function in the equipment used to receive and decode the information. Depending upon which of the two control characters has been sent last, the receiving equipment interprets each encoded character in two alternative ways.

If the "letters" control character has been sent last, the encoded characters which follow are assumed to be alphabetic characters. If on the other hand the "figures" control character has been sent last, the characters which follow are assumed to be either numerals, punctuation marks, or non-printing characters.

In other words, the Baudot code is not really a 5-bit code, but is equivalent to a 6-bit code. The information which would otherwise be carried by a sixth bit is sent only when required, in the form of the two control characters "figures" and "letters". This is an economical approach, and fairly satisfactory for many purposes, although it can be rather unwieldly where the information to be handled contains a lot of mixed alphabetic and numeric characters.

Chapter 8

The flipflop family

Apart from logic gates probably the most basic elements in digital circuits are flipflops, which form the basis of sequential logic and storage systems. In this chapter we look at the operation of simple two-gate flipflops, and at the various types of practical flipflop which have been derived from them.

The digital circuit elements we have considered in the foregoing chapters have performed functions which are substantially independent of time. For example the output level of an OR gate will always be true if at least one of its inputs is true, and false only if all inputs are false. This relationship between inputs and output is not dependent upon time, except in a second-order sense: when the input situation changes significantly, it can take a few tens of nanoseconds for the change to propagate through to the output.

In contrast with these circuit elements are others whose function does depend quite significantly on time. The most common of these are the various forms of flipflop, and the monostable or "one-shot". These and one or two other time-dependent circuit elements form the basis of sequential logic circuits, information storage systems, binary arithmetic and counting circuits, and a host of other valuable applications of digital electronics.

For the present, let us turn our attention to flipflops, which are probably the most important of all time-dependent digital elements.

As the name "flipflop" suggests, the most obvious characteristic of these devices is that they are bi-stable. They have two stable operating conditions, and can be made to switch from one to the other.

A very basic flipflop can be formed by connecting two 2-input NAND gates, as shown in Fig. 1. As you can see, the two gates are "cross coupled", with the output of each being connected to one of the two inputs of the other. This forms a regenerative feedback loop, with the result that if the two uncommitted inputs are taken to the true (1) logic level, only one of the gates can have a true output; the other must have a false (0) output.

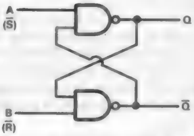
This happens because the output that is true causes both inputs of the other gate to be true, driving the second gate's output false by virtue of the inherent inversion in a NAND element.

Only one output of the flipflop (or "FF") is normally true, then, and the other is false. The two outputs are logically complementary, in other words—each is the logical complement of the other. By convention one output is usually labelled "Q", and the other "Q-bar".

This labelling is quite arbitrary, as the FF is basically a symmetrical device. Normally we are quite free to call either output "Q", as convenient, with the other becoming Q-bar by implication.

The FF has two stable "states", then: one with Q true and Q-bar false, and the other with Q false and Q-bar true. Just which of the two states it will be in at any particular time depends upon its previous history.

For example on each occasion when power is initially applied to the circuit, the FF will always tend to "come on" in the same state. This is because there are always small differences in the characteristics of any two gates, so that one gate will always tend to "lead" when power is first applied—steering the FF in the resulting direction. It might always tend to come on with Q true, or alternatively with Q-bar true.



A	B	BEFORE		AFTER	
		Q	$\overline{Q}$	Q	$\overline{Q}$
1	1	0	1	0	1
1	1	1	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
0	0	0	1	?	?
0	0	1	0	?	?

FIG. 1

The other main factor which determines the state of the FF is any signals which may be applied, or may have been applied, to the inputs A and B. There are various possibilities here: the FF may have either Q or Q-bar true initially, while either A or B may be taken true or false independently. With three variables, this gives eight different situations, each of which can be analysed using the basic principles of gate operation.

You can try doing this if you like, as it would be a good exercise. However, the results are shown in concise form in the truth table, and it may be sufficient just to consider these carefully. Each pair of lines covers one of the four possible truth-value combinations for the two inputs, with the two lines in each pair covering the two possible initial states of the flipflop.

The first thing to notice is that while ever both the A and B inputs are held true (1), the FF outputs do not alter. This is shown in lines 1 and 2, where you can see that the FF simply remains stable in whichever of the two states it was initially.

The next thing to notice is that if input A alone is taken to the false logic level (0), the FF always ends up in the state where Q is true and Q-bar is false. If it is initially in this state, it simply remains there (line 4), but if it is initially in the opposite state it will promptly change over or "toggle" (line 3).

Conversely, if input B alone is taken to the false logic level, the FF always ends up in the state where Q is false and Q-bar is true. Again if it is initially in this state it merely remains there (line 5), while if it is initially in the opposite state it will toggle (line 6).

By convention it is usual to call the state of the FF where the Q output is true the "set" state, and the alternative state where Q is false the "reset" or "clear" state. So that from the table in Fig. 1 we can see that a false logic level applied to the A input causes the FF to always end up in the set state, while a false logic level applied to the B input causes the FF to always end up in the reset state.

Because of this the A input could be labelled S-bar, as shown in brackets. The "S" stands for "set", indicating that the input may be used to force the FF into the set state, while the bar indicates that a false logic level produces this action rather than a true logic level.

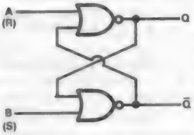
Similarly the B input could be labelled R-bar, as shown in brackets, with the "R" standing for "reset" to indicate that the input may be used to force the FF into the reset state. Again the bar indicates that the input is effective when taken to the false logic level, rather than the true level. An alternative to labelling this input "R-bar" is to label it "C-bar", with the C standing for "clear".

The last pair of possibilities for the FF of Fig. 1 is when both inputs are taken false at the same time. As you can see from the table, this results in an "unpredictable" situation—symbolised by the question marks (lines 7 and 8).

The reason for this is that when both inputs are taken to the false level at the same time, both gates are forced into the condition where their outputs go true. The FF is thus forced into a third state, which corresponds to neither of its two normal stable states. While it will remain in this third state as long as both inputs are held false, it will revert to one of the two stable states as soon as either of the inputs is taken true.

Which of the two states it reverts to depends upon which of the two inputs is returned true first. If B is taken true first, it will revert to the set state, while if A is returned true first it will revert to the reset state.

What if both inputs are returned true simultaneously? Assuming this is possible (and it is easier said than done), then a "race" condition occurs. Both gates will attempt to drive their outputs low, and the outcome will depend upon subtle differences between them in terms of speed, etc.—rather like the situation when power is first applied.



A	B	BEFORE		AFTER	
		Q	Q̄	Q	Q̄
1	1	0	1	?	?
1	1	1	0	?	?
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
0	0	0	1	0	1
0	0	1	0	1	0

FIG. 2

Because of these complications, the simple type of flipflop shown in Fig. 1 is never used in situations where both inputs are likely to be taken to the false logic level and returned true simultaneously. Its use is restricted to situations where a FF is only required to act in response to single false-going input pulses, applied to the S-bar and R-bar inputs in an exclusive manner.

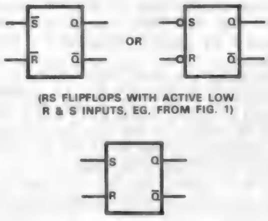
Because the simple type of FF shown in Fig. 1 is only capable of switching back and forth between its set and reset states, it is usually called a "reset-set" or **RS flipflop**.

An RS flipflop can also be made using two NOR gates, as shown in Fig. 2. The method of cross-coupling the gates is virtually identical to that used with NAND gates, and the operation turns out to be rather similar. However, there are subtle differences, as the truth table shows.

Perhaps the first difference to note is that in this case the "no change" situation occurs when both inputs are held at the false logic level, rather than the true level. This arises from the nature of NOR gates, compared with NAND gates.

For the same reason, the "indeterminate outcome" situation now occurs when both inputs are taken to the true logic level. As before this results in a potential "race" condition, if both inputs are returned to the inactive logic level (here 0) simultaneously.

Superficially it may seem that the two remaining " one input alone" situations are unchanged from the corresponding situations for Fig. 1. However this is not really so. The difference is that here the inputs are effective or "active" when they are taken true, rather than when they are taken false as in Fig. 1.



(RS FLIPFLOPS WITH ACTIVE LOW R & S INPUTS, EG. FROM FIG. 1)

(RS FLIPFLOPS WITH ACTIVE HIGH R & S INPUTS, EG. FROM FIG. 2)

FIG. 3 R-S FLIPFLOPS

In other words the A input of Fig. 2 acts as a reset or "R" input, the absence of a complementing bar showing that it is active when taken to the true logic level. Similarly the B input now acts as a set or "S" input.

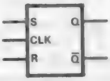
Although simple RS flipflops may be made by cross-coupling gates as shown in Figs. 1 and 2, and this is often done, they are also made available by IC manufacturers as ready-made building blocks. In this form, they are usually represented by the symbols shown in Fig. 3.

The upper symbols show an RS flipflop with active low inputs, like that of Fig. 1 when the positive logic convention is used. Note that there are two possible symbols, one with the inputs labelled S-bar and R-bar, and the other with them labelled S and R but using "bubbles" to indicate that they are active low.

The lower symbol shows an RS flipflop with active high inputs, like that of Fig. 2 when positive logic convention is again adopted.

It is worth noting that an RS flipflop is in fact an elementary "memory" or storage cell. It is able to store one bit (binary digit) of information, whose value is 1 in the set state or 0 in the reset state. The information may be stored in the cell by feeding in pulses at the inputs, with the S input used to store a 1 and the R input to store a 0. If you like, the FF "remembers" which input last provided a pulse.

While a simple RS flipflop has its uses, there are many applications in digital systems where flipflops are required to respond only at certain fixed times, as determined by general timing or "clock" signals which are fed throughout the system. To provide for this sort of operation, a number of variations on the basic RS flipflop have been evolved.

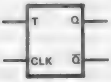


S	R	BEFORE CLK PULSE		AFTER CLK PULSE	
		Q	Q̄	Q	Q̄
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	?	?
1	1	1	0	?	?

FIG. 4 CLOCKED OR GATED RS FLIPFLOP

Probably the simplest of these is the **clocked-RS flipflop**, which is shown in Fig. 4. This looks rather similar to the basic RS flipflop, having S and R inputs as before; but there is a difference. Changes in logic level at the S and R inputs now do not cause immediate changes in the state of the FF, they merely condition it to respond in a certain way upon the arrival of a timing pulse at the CLK or clock input.

As you can see from the truth table, the response of the FF when the clock pulse arrives closely parallels that of the basic RS flipflop. If both S and R inputs are at the false logic level when the pulse arrives, the FF state remains unaltered. If the R input alone is held at the true level, the FF will either remain in or switch to the reset state when the clock pulse arrives. Similarly if the S input alone is held at the true level, the arrival of the



T	BEFORE CLK PULSE		AFTER CLK PULSE	
	Q	Q̄	Q	Q̄
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

FIG. 5 T-TYPE OR TOGGLE FLIPFLOP

clock pulse causes the FF to either remain in or switch to the set state.

If both the S and R inputs are held true when the clock pulse arrives, there is again an "indeterminate" outcome. Following the clock pulse the FF might be left either set or reset, depending upon any internal speed-of-response bias it may possess. In this respect the clocked-RS flipflop is similar to the basic RS type—except that here the indeterminate situation generally only arises if the S and R inputs are held true simultaneously with the arrival of the clock pulse. In most cases they can both be true at other times without producing this result.

Another type of clocked flipflop you may encounter is the so-called **T-type or "toggle" flipflop**, shown in Fig. 5. Like the clocked-RS flipflop this has a CLK input, but in place of the R and S inputs it has a single input labelled "T". This has a controlling effect as shown in the truth table.

As you can see, when the T input is held false during the clock pulse this has the effect of "freezing" the FF in its initial state; the state after the clock pulse is the same as that before. However, if the T input is held true during the clock pulse, the clock pulse forces the FF to "toggle" or change states—regardless of its initial state. If it was set, it will reset; and vice-versa.

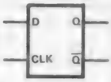
The T-type FF is thus capable of only two responses to a clock pulse. It can either remain unchanged, or toggle, depending upon the logic level applied to the T input. Unlike the clocked-RS FF there is no "indeterminate" response.

A further type of clocked flipflop is the so-called **D-type flipflop or "latch"**, shown in Fig. 6. Instead of the T input this has a "data" or "D" input, whose controlling action is shown in the truth table.

If the D input is held false when the clock pulse is applied to the FF, its effect is to cause the FF to either remain in, or switch to the reset state. Conversely if the D input is held true, the arrival of the clock pulse causes the FF to either remain in the set state, or switch to it.

In other words, the D-type FF acts rather like a clocked-RS FF which has an S input and an R-bar input, tied together internally. When the D input is high it causes the FF to be set, like an S input, and when it is low it causes the FF to be reset, like an R-bar input.

The reason for calling the D-type FF a "latch" is that when the clock pulse arrives, it effectively stores whatever the logic level happens to be at the D input—"latching" it in storage until the arrival of the next clock pulse.



D	BEFORE CLK PULSE		AFTER CLK PULSE	
	Q	Q̄	Q	Q̄
0	0	1	0	1
0	1	0	0	1
1	0	1	1	0
1	1	0	1	0

FIG. 6 D-TYPE FLIPFLOP OR LATCH

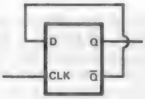


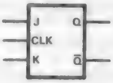
FIG. 7 D-TYPE FLIPFLOP CONNECTED FOR TOGGLING

Note that like the T-type FF, the D-type FF always operates in a determined fashion; there is no "indeterminate" situation, where the outcome cannot be predicted.

Incidentally, the D-type FF can be used as a toggling FF simply by connecting its D input back to its Q-bar output, as shown in Fig. 7. This forces the FF to toggle to its oposite state each time a clock pulse arrives, like a T-type FF with its T input held true. You might care to verify this yourself by checking through the truth table.

Probably the most important type of clocked FF is the so-called **JK flipflop**, which is shown in Fig. 8. This is a very flexible element, in that it can be arranged to perform virtually all of the functions performed by the other types.

The JK flipflop has two main inputs in addition to the CLK input, and as you can see these are labelled "J" and "K". As shown in the truth table, these function rather like the S and R inputs of a clocked-RS flipflop. But there is one important difference—when both J and K inputs are held true during the clock pulse, the outcome is no longer indeterminate. The FF simply toggles, like a T-type FF with its T input held true.



J	K	Q (BEFORE)	Q (AFTER)
0	0	0	0
0	0	1	1
1	0	0	1
1	0	1	1
0	1	0	0
0	1	1	0
1	1	0	1
1	1	1	0

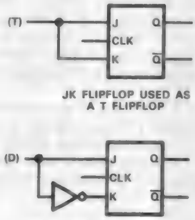
FIG. 8 JK FLIPFLOP

In a sense, then, the JK flipflop is virtually an "improved" version of the clocked-RS flipflop, designed so that its operation is always completely predictable. If the J and K inputs are held false, it will remain unchanged; if the J input alone is held true, it will set; if the K input alone is held true, it will reset; and if both J and K are held true, it will toggle.

JK flipflops are used in large numbers in most digital circuits and systems, because of their flexibility. As an example of this , Fig. 9 shows how JK flipflops may be used to perform the same functions as a T-type FF (upper diagram) and as a D-type FF (lower diagram). In one case the J and K inputs are simply tied together, while in the other case the K input is fed through an inverter whose input is tied to the J input, so that it always receives the complement of the logic level applied to the J input. Or if you like, the inverter effectively changes the K input into a K-bar input.

You might care to verify for yourself that these connections cause the JK flipflop to act like the T-type and D-type flipflops, using the truth table of Fig. 8 for reference. This would help to reinforce the concepts in your mind.

Practical JK flipflops quite often have other inputs in addition to the basic J, K and CLK inputs. Some of these are illustrated in Fig. 10—but note that not all of the inputs shown would necessarily be provided on any particular device.



JK FLIPFLOP USED AS A T FLIPFLOP

FIG. 9 JK FLIPFLOP USED AS A D FLIPFLOP

In addition to the J and K gating inputs, there may also be provided J-bar and K-bar inputs which are active for the opposite logic polarity. These may either be labelled J-bar and K-bar, or have the usual J and K labels but with "bubbles" to indicate the active-low function for the positive logic convention. The diagram shows the latter approach.

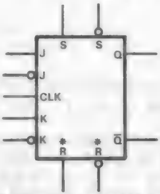
The flipflop may also be provided with direct-acting S and R inputs, quite separate from the inputs concerned with clocked operation. These may again be either active-high or active-low, with the latter labelled either S-bar and R-bar (or C-bar) or S and R (or C) with bubbles. Direct-acting S and R inputs enable a JK flipflop to be preset to the set or reset state, as required, before its clocked operation takes place.

There is one further type of flipflop which you may encounter sometimes in digital circuits. This is again a simple RS flipflop, but one made up from two inverters as shown in Fig. 11.

Having two inverting logic elements cross-coupled, this circuit has two stable states like the RS flipflops shown in Figs. 1 and 2. But note that because it uses inverters, its outputs are in fact identical with its inputs.

To make this very basic FF change its state, one of the input-output terminals must be "forced" briefly to the opposite logic level, against the cross-connected output. Generally this causes no harm, because almost immediately after the input reaches the "other" logic level, the FF changes state to maintain the new state of affairs.

Unlike the RS flipflops in Figs. 1 and 2, however, the inputs of Fig. 11 do not become "inactive" when they have triggered the FF to switch to its opposite state. Both input-outputs are active here at both logic levels, so that the FF is quite capable of following rapid alternations of logic level. This means that care must be taken if a simple two-inverter FF is used for such applications as "debouncing" signals derived from mechanical switch contacts. We will look at this further in a later chapter.



*MAY BE CALLED CLEAR (C)

FIG. 10 POSSIBLE CONNECTIONS ON A PRACTICAL JK FLIPFLOP

Although we have looked at a number of different flipflops in this chapter, this has been largely for the sake of completeness. It isn't really necessary to memorise the detailed operation of every one, as the main ones you are likely to encounter are the simple RS type (Figs. 1, 2 and 11) and the JK type (Figs. 8, 9 and 10).

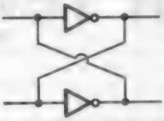
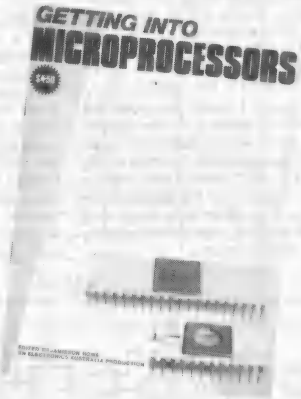


FIG. 11 RS FLIPFLOP FROM TWO INVERTERS

An important thing to bear in mind about flipflops in general is that they are usually quite symmetrical devices, so that most of the labels applied to their inputs and outputs are arbitrary. This means that if it is convenient to do so, there is generally no reason why you can't swap the labels around. For example you could swap the Q and Q-bar outputs of a JK flipflop as in Fig. 8, providing you also swap the labels for the J and K inputs to match. If there are also S and R inputs, these would have to be swapped as well.

The only thing to watch when this is done is that you don't forget any inputs and outputs which are not one of a pair. Some of these will change into their logical complement, while others won't change at all. For example a CLK input doesn't change, nor does a T input; but a D input will change into a D-bar input. Similarly if the FF only has a nominal R input, it will become an S input; and if it only has a nominal Q output, this will become a Q-bar output.

Finally, a word on packaging. Flipflops tend to come both singly and in multiples. If you want an elaborate JK type, with the full complement of inputs, you'll probably find they come only one to an IC package. But if you need only simple JK types, you may well find that suitable types come as packages of two, or even four. And if you have an application requiring a number of basic T-type or D-type FFs, you may be able to get these in packages of six or eight together.



Chapter 9

Flipflops in registers

Probably the most basic use of flipflops in digital circuits is to form registers, for storing and manipulating binary data. In this chapter we look at basic shift and latch registers, and also at more complex registers capable of performing multiple functions.

As we saw in the preceding chapter, flipflops are capable of storing information. A single flipflop is capable of storing one binary digit or "bit", which is the elementary unit of information. By using a number of flipflops in combination, it is possible to store the same number of bits of information.

By convention, any group of devices used to store and manipulate information is called a "register". Registers may be made up using virtually any sort of information storage device, including such things as mechanical switches and relays. However, most of the registers found in modern digital circuits are formed using flipflops as the storage elements. A group of flipflops used for information storage and manipulation is thus described as a flipflop register.

Probably the most basic type of flipflop register is the shift register. In its simplest form, this consists of a group of D-type flipflops connected as in Fig.1. As you can see, the clock inputs are all connected together to form a common "clock line", while the Q output of each flipflop is connected to the D input of the flipflop which succeeds it moving to the right.

You will recall, I hope, that upon the arrival of a clock pulse a D-type flipflop adopts the state which makes the logic level at its Q output match the level which was present at its D input. Thus, when a clock pulse is applied to the arrangement shown in Fig.1, its effect is to cause each flipflop to adopt the state corresponding to the level at its D input.

This means that FF4 will adopt the state previously held by FF3, because its D input connects to the Q output of FF3. Similarly FF3 will adopt the state previously held by FF2, and FF2 will adopt the state previously held by FF1. And FF1 will adopt the state which corresponds to whatever logic level is present at its D input.

In other words, the effect of the clock pulse is to cause the register flipflops to "shuffle" or shift their stored bits of information one place to the right. Hence the name "shift register".

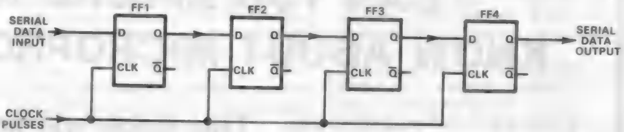


FIG. 1 : BASIC 4-BIT SHIFT REGISTER

It should be fairly obvious that we can feed a 4-bit binary number into such a shift register, by applying its four bits in sequence to the D input of FF1, and applying a clock pulse each time. The first clock pulse will cause the first data bit to be stored in FF1, while the second clock pulse will cause the first data bit to be shift into FF2, and the second data bit to be entered into FF1. Similarly the third clock pulse will cause FF3 to receive the first data bit, FF2 the second data bit, and FF1 the third data bit. Finally the fourth clock pulse will cause FF4 to receive the first data bit, FF3 the second data bit, FF2 the third data bit and FF1 the fourth data bit.

We can store the 4-bit number in the register as long as we like, simply by applying no further clock pulses— assuming we also maintain the flipflop power supply, of course.

When we wish to extract the number from the register, we can shift it out again by applying four more clock pulses. The four data bits will then appear in sequence at the Q output of FF4.

The D input of FF1 thus acts as a serial data input for the register, while the Q output of FF4 acts as a serial data output.

Simple serial-in/serial-out shift registers like that of Fig.1 are made prepackaged in a variety of sizes, with different numbers of flipflops fabricated in a single IC package. Typical devices have from 4 flipflops as shown, to as many as 4096 flipflops. They are used quite frequently in digital circuits, mainly for binary data storage.

They are also used as time delay elements, by making use of the fact that if binary data is applied to the serial input with continuous clocking, the data bits take an appropriate number of clock pulse periods to shift through the register and reappear at the serial output. For the 4-bit register shown, data will thus take 4 clock periods to shift through the register.

In general terms, a serial-in/serial-out shift register with "N" flipflops (where N is any integral number) may be used to delay binary data by N clock pulse periods.

Fairly obviously, the basic shift register of Fig.1 is only capable of shifting data in one direction. At times, it can be very desirable to have a register capable of shifting data in either direction. For these applications a slightly more complex shift register is used, with logic gates to allow control of the shifting direction. This is illustrated in Fig. 2.

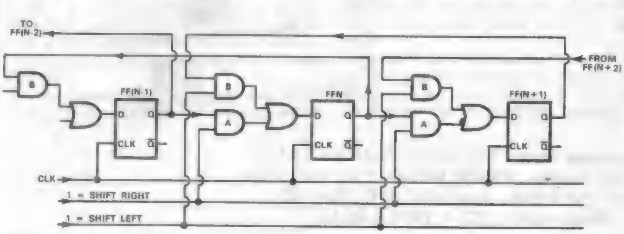


FIG. 2 : SECTION OF A BIDIRECTIONAL SHIFT REGISTER

Basically the gating allows the D input

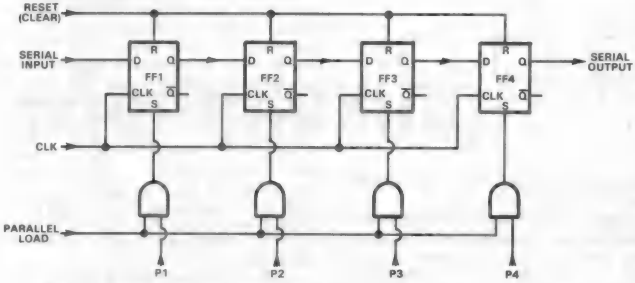


FIG. 3 : 4-BIT SHIFT REGISTER WITH CLEAR AND PARALLEL LOADING

of each FF in the register to be connected either to the Q output of its neighbour to the left, as before (via the AND gates marked "A"), or to the Q output of its neighbour to the right (via the AND gates marked "B"). The two AND gates associated with each input connect to it via an OR gate, as shown.

The control inputs of all the "A" AND gates are connected together, as shown, as are the control inputs of the "B" AND gates. Thus by applying a logical 1 to the control line for the "A" gates, the register can be set up for shifting data to the right. Alternatively a logical 1 can be applied to the control line for the "B" gates, and this will set up the register for left shifting.

It would be usual to label the "A" gate control line as a "shift right enable" line, signifying that its function is to enable the shift right capability of the register. Similarly the "B" gate control line would be labelled "shift left enable".

In the shift registers of Figs. 1 and 2, the only way of getting data into the register is serially—ie., one bit at a time. This can be too slow in many applications, and accordingly many registers are provided with a facility to load data in parallel fashion (all bits simultaneously). One way of doing this is shown in Fig.3.

Each flipflop of the register is provided with direct set or S inputs, as you can see, and each S input is fed by an AND gate. One of the two inputs of each AND gate is connected to a common control line, while the remaining inputs are kept separate and become the parallel inputs of the register—P1, P2, P3 and P4.

To load the data present at the parallel inputs into the register, a short enabling pulse is applied to the gate control line (in other words, the line is taken briefly to the logic 1 level). Those gates whose data inputs are at logic 1 level will therefore be enabled, and their outputs will accordingly go to logic 1 level also. This will cause their associated FFs to be forced into the set state, if they were not previously in that state. If they were already in the set state, they will remain there. Either way, the logic 1's present at the data inputs effectively will have been loaded into the FFs.

Note, however, that when the gate control line is pulsed, only those gates whose data inputs are at logic 1 level will be enabled. The other gates will not be enabled, because their data inputs will be at logic 0. As a result the flipflops associated with these gates will not be affected—even if they happen to be set to a 1, which is the exact opposite of the data bit we are trying to load in.

Because of this problem, the flipflops of Fig.3 are provided with direct reset or "clear" inputs as well, and these are connected together to form a common reset or clear line. This is used to reset all of the FFs together, before the actual parallel loading is performed.

This means that just before the input gate control line is pulsed, all of the flipflops in the register are forced into the reset state, wherein they contain 0's. Hence when the gate control line is pulsed and the gates whose data inputs are at logical 1 level force their associated FFs into the set state, the remaining FFs will all remain in the reset state—effectively storing the 0's present at the data inputs of their gates, even though these gates actually remain inactive.

So that with a register provided with the simple type of parallel data loading scheme shown in Fig.3, parallel loading actually becomes a two-step process. First the reset or clear line is pulsed, to clear the register, and then the parallel load enable line is pulsed to allow the input gates to enter the "1" data bits as appropriate.

While this form of parallel data loading is adequate for many purposes, there are situations where the required two-step process can be inconvenient. There are also other situations where it is required to perform parallel loading of data synchronously—ie., in step with the system clock pulses.

To cope with such situations, another type of parallel data input loading can be used. This is known as "jam transfer", and as the name suggests it allows parallel data to be jammed into the register flipflops regardless of the data bit values or the previous states of the flipflops. And the transfer takes place in response to a normal clock pulse, so that it meets the synchronism requirement.

One way of achieving jam transfer parallel loading is shown in Fig.4. As you can see the D input of each FF is provided with a gating system rather like that of Fig.2, with two AND gates feeding each input via an OR gate. And as before, the "A" gates are used to connect the D inputs to the Q-output of the preceding FF (or the serial input, in the case of FF1), for right shifting.

Here the other set of AND gates (labelled "B") are used not to set up the data paths for left shifting, but to enter the parallel data. The data input of each gate becomes the parallel input for its associated flipflop. So that when a clock pulse is applied to the register clock line, data bits from the parallel inputs P1, P2, P3 . . . will be forced into the flipflops if the "B" gates have been enabled by a logic 1 level applied to their control inputs.

Note that this will be a true jam transfer, because when a D-type FF is clocked, it is forced to adopt the state corresponding to the logic level present at its D input. In other words, we have a jam transfer here because a flipflop D input is active at both logic levels, whereas the S inputs used in Fig.3 are active only when taken to the true logic level.

Quite incidentally, Fig.4 shows the control lines for the two sets of gates interconnected via an inverter. This is a technique often used, with the idea of reducing the number of separate control signals required to control register operation. With the scheme shown, a logic 1 level applied to the "Mode" input causes the "A" gates to be enabled, so that the register performs the shift right operation. However if the logic level at the Mode input is changed to an 0 this not only dis-

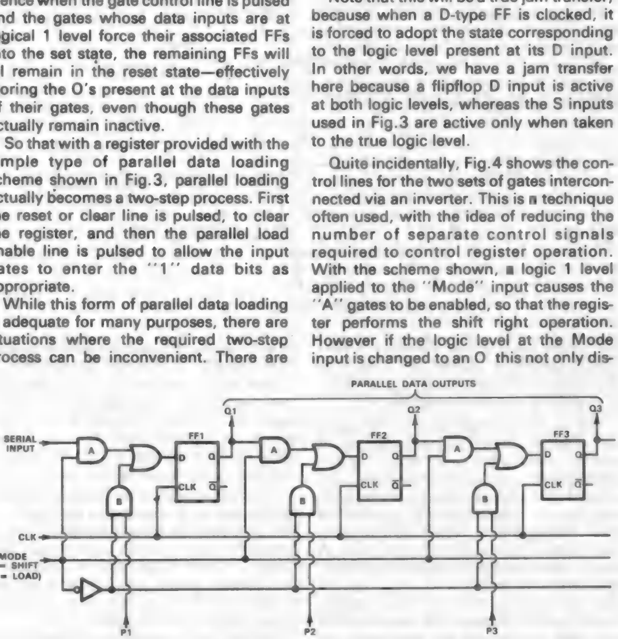


FIG. 4 SECTION OF A SHIFT REGISTER WITH JAM-TRANSFER PARALLEL LOADING

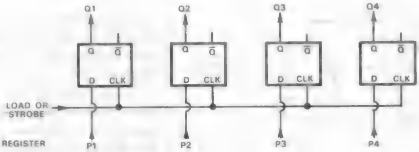


FIG 5 : 4-BIT LATCH REGISTER

ables the "A" gates but also enables the "B" gates via the inverter—so that the register performs the parallel loading operation. A single control signal is thus able to determine the register's mode of operation.

The flipflops in Fig.4 are also shown with their Q outputs brought out, with labels Q1, Q2, Q3, etc. This is done with some shift registers, as it allows a further degree of flexibility: data bits can be taken from the register in parallel, as well as being loaded in parallel.

One important use for registers fitted with parallel data inputs or outputs is serial-parallel and parallel-serial conversion. For example, a binary number available only in parallel form may be converted into serial form by loading it into a register via the parallel inputs, and then shifting it out bit by bit.

Conversely a binary number available only in serial form may be converted into parallel form by shifting it into a register via the serial input, and then sensing the bits simultaneously at the parallel outputs.

Some pre-packaged flipflop registers are provided with parallel inputs and a serial output, expressly for parallel-to-serial conversion. Others are provided with parallel outputs and a serial input, expressly for serial-to-parallel conversion.

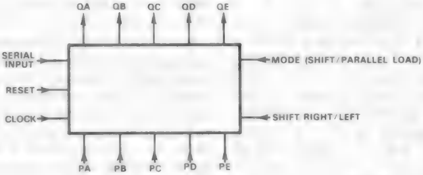


FIG 7 : MULTI-PURPOSE 5-BIT REGISTER

Still others are provided with all of these, and are therefore capable of performing both types of conversion quite apart from pure shifting and data storage.

Not all flipflop registers are shift registers, though. In fact one simple type of register quite often encountered is the latch register, which is used purely for parallel data storage. As you can see from Fig.5, it consists essentially of nothing more than a row of D-type flipflops with their clock inputs connected together. A clock pulse applied to the common clock line effectively causes each FF to load in the data bit present at its D input, so that the D inputs become the parallel data inputs of the register and the clock line becomes a "load", "store" or "strobe" control line.

While Fig.5 shows a latch register for only four bits, pre-packaged latch register ICs are available which provide for up to eight or ten bits. And of course longer latch registers may be assembled if required, using multiple devices. This also applies to registers of the type shown in Figs. 3 and 4.

The registers we have looked at so far in this chapter have been only capable of shifting data right or left, parallel loading, or clearing. While these are the operations most commonly performed by registers, other operations are at times required, and accordingly some registers are provided with the appropriate facilities.

One such facility is the ability to complement the stored data bits—in other words, each FF of the register may be toggled to its opposite state. This can be of value where a register is used for binary arithmetic, as we will see later on.

To provide such a bit complementing facility, gating is provided so that application of a suitable control signal causes a data path to be opened up between the D input of each FF and its own Q-bar output. A clock pulser then causes each FF to adopt the state formerly present at its Q-bar output—ie., it toggles.

Another facility which may be provided is the ability to "rotate" the bits of the stored data, either to the right or the left. This is similar to shifting, except that a data path is opened between the FFs at the two ends of the register so that none of the bits are "lost".

To rotate data to the right, for example, the Q output of the FF at the right-hand end of the register is connected via gating to the D input of the FF at the left-hand end, while all of the other D inputs are connected as in Fig.1 for right shifting. A clock pulse then causes all of the data bits to be shifted to the right, except for the rightmost bit which is rotated back into the leftmost FF.

Similarly, by using gating to connect the D input of the rightmost FF of a register to Q output of the leftmost FF, in addition to connecting all the other data paths for left shifting, it becomes possible to perform a "rotate left" function.

Other facilities which may be provided on registers include parallel logic functions such as AND, OR, and exclusive-OR. These are found somewhat less frequently, but have important uses in data processing. Broadly speaking they are provided rather like jam-transfer parallel loading, except that additional gating is used to provide the required logic.

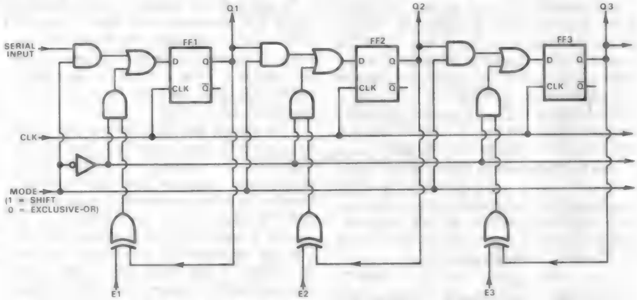


FIG 6 SECTION OF A SHIFT REGISTER WITH EXCLUSIVE-OR PARALLEL LOADING

This is illustrated in Fig.6, which shows a basic shift-right register provided with an alternative exclusive-OR facility. As you can see, it is broadly similar to Fig.4. The difference is that instead of being directly applied to the D inputs of the FFs when the Mode line is taken to logic 0, the parallel data inputs are fed via exclusive-OR gates where they are compared with the existing data bits in the FFs. This is done by connecting one input of the ex-OR gates to the FF Q outputs.

When registers are implemented using single ICs, it is not usual to show all of the internal FFs and gating on a logic diagram. Thus a single-IC register of 5-bit capacity may be shown by a symbol like that in Fig.7, with a simple rectangle representing the register as a whole. All that is shown are the various control signal inputs, together with the data inputs and outputs. This is generally quite adequate for design purposes, providing the functions of the various inputs and outputs are clearly defined.

Chapter 10

Flipflops in counters

Apart from registers, the most common use for flipflops in digital circuits is in counters and frequency dividers. In this chapter we look at the various types of counter circuit, including ripple-carry, synchronous, up-down, ring and Johnson counters.

In many digital systems, there is a need to count the number of pulses which may occur at a given point. Quite a number of different circuits have been developed to perform this job, all of them generally given the title "counter". Nowadays just about all counters are based on flipflops, connected up in various configurations.

Before we look at the most common counter configurations, it is worth noting the difference between counting and two similar but slightly different operations: scaling, and frequency division.

When a circuit is used as a counter, we interpret its "output" as a number—generally a binary or BCD number—which continuously corresponds to the number of input pulses received by the counter input after the circuit has been reset or cleared. We expect to be able to examine the count at any time we choose, to see how many pulses have occurred.

There are times, however, when we don't really need to know the count on a continuous basis. In fact it is sometimes sufficient to have nothing more than a guide to the rate at which input pulses are being received. This can be done by arranging for the circuit to simply deliver an output pulse each time a certain number of input pulses are received—say every 10, every 100 or every 1000. A circuit used in this way is said to be "scaling" rather than counting, and would be described as a "scaler".

If a circuit capable of performing scaling is fed with a regular stream of input pulses—i.e., pulses having a fixed repetition rate—it will deliver output pulses which will also have a fixed repetition rate. But the output rate or frequency will be equal to the input rate or frequency divided by the scaling factor. In other words, the circuit will actually be performing frequency division.

Frequency division is merely a special type of scaling, if you like, where the input and output signals have a fixed repetition rate.

Most of the counter circuits we will be discussing in this chapter are also capable of performing scaling or frequency division, as required.

One more general point. It is usual to classify counter circuits according to the number of input pulses which they are capable of receiving before effectively returning to their initial state. This is known as their "modulo factor". A counter which counts 10 pulses before returning to its initial state is thus described as a "modulo-10" counter, while one which counts to 16 before returning to its initial state is described as a "modulo-16" counter.

Another way of defining the modulo factor of a counter is to say that it describes the number of discrete counting states it provides. So that a modulo-16 counter provides 16 counting states, a modulo-10 counter 10 states, and so on. But note that the "clear" or all-zeros state is regarded as one of these states, and counted if it occurs.

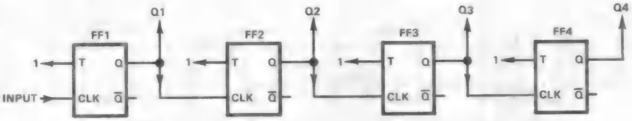


FIG. 1 : RIPPLE-CARRY BINARY COUNTER-MODULO 16

The modulo factor of a circuit when used as a counter is numerically equal to its scaling factor when used as a scaler, and its division ratio when used as a frequency divider. So that a modulo-16 counter may be used as a x16 scaler or a divide-by-16 frequency divider, and so on.

The simplest type of counter circuit is formed by connecting a number of T-type flipflops together as shown in Fig. 1. The input pulses are fed to the clock input of the first flipflop FF1, while the clock input of each successive flipflop is taken from the Q output of the preceding flipflop.

Fairly obviously, FF1 will change state upon the arrival of each input pulse, as they are fed directly to its clock input. However because the clock input of FF2 is fed from the Q output of FF1, it will receive a complete "input pulse" only when FF1 completes a full reset-set-reset sequence. FF2 will therefore change state on every second input pulse.

Similarly because the clock input of FF3 is fed from the Q output of FF2, it will change state only for every fourth input pulse. And FF4 in turn will change state only for every eighth input pulse. The logic levels of the four flipflops for a sequence of input pulses will therefore be as shown in the truth table.

As you can see, the four flipflops effectively count the input pulses in binary fashion, with FF1 counting the units, FF2 the twos, FF3 the fours, and FF4 the eights. As a group the four flipflops are able to count up to 16 pulses before repeating the counting sequence. Hence this simple circuit may be described as a binary modulo-16 counter.

Because a flipflop takes a finite time to change state after being triggered at its clock input, a counting circuit like that shown in Fig. 1 does not respond to an input pulse by immediately adopting the new count. If FF2 is required to change, it cannot do so until FF1 has changed its state; similarly if FF3 is to change, it cannot do so until FF2 has changed; and so on. The changes which take place following the arrival of an input pulse therefore tend to "ripple" down the counting chain, with the last FF tending to change somewhat after the first.

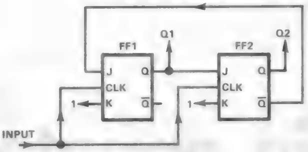
INPUT PULSES	Q1	Q2	Q3	Q4
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	1	0	1	0
6	0	1	1	0
7	1	1	1	0
8	0	0	0	1
9	1	0	0	1
10	0	1	0	1
11	1	1	0	1
12	0	0	1	1
13	1	0	1	1
14	0	1	1	1
15	1	1	1	1
16	0	0	0	0

Counters using the simple scheme illustrated in Fig. 1 are therefore known as "ripple-carry" counters.

A simple ripple-carry binary counter of this type can be made with any desired number of flipflops, and will count to the corresponding power of 2. In general, a chain of "N" flipflops connected as in Fig. 1 will count up to the Nth power of 2. So that five flipflops will count to 32, six will count to 64, seven will count to 128, and so on.

While simple counters of this type are often used in digital circuits, there are situations where it is required to have a counter with a modulo factor which does not correspond to a power of two. One way of providing such counters is to start with a simple binary counter which has sufficient flipflops to provide the first binary modulo numerically larger than the required modulo. Then some form of feedback or gating is applied, so that the counter effectively "skips" some of the normal counting states, to provide the required modulo.

By using JK flipflops, the required gating can often be provided by the flipflops themselves. This is illustrated by the simple example in Fig. 2, which shows two flipflops connected to form a modulo-3 counter. Note the connection between the Q-bar output of FF2 and the J input of FF1, which forces FF1 to remain reset when FF2 is set.



INPUT PULSES	Q1	Q2
0	0	0
1	1	0
2	0	1
3(0)	0	0

FIG. 2 : MODULO-3 COUNTER

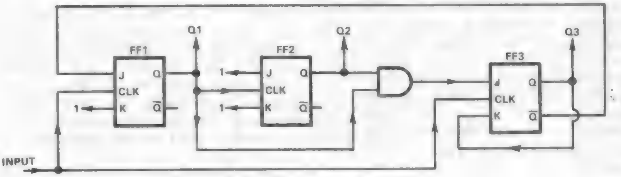


FIG. 3 : MODULO-5 COUNTER

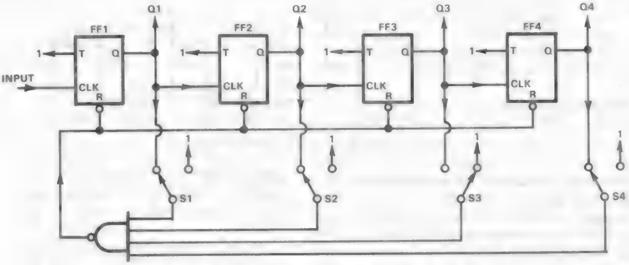


FIG. 4 : ADJUSTABLE-MODULO COUNTER (SET FOR MODULO-11)

INPUT PULSES	Q1	Q2	Q3	Q4
0	0	0	0	0
1	1	1	1	1
2	0	1	1	1
3	1	0	1	1
4	0	0	1	1
5	1	1	0	1
6	0	1	0	1
7	1	0	0	1
8	0	0	0	1
9	1	1	1	0
10	0	1	1	0
11	1	0	1	0
12	0	0	1	0
13	1	1	0	0
14	0	1	0	0
15	1	0	0	0
16(0)	0	0	0	0

A further example of the way gating may be used to provide counters with non-binary modulos is shown in Fig. 3. Here three flipflops are used, connected to perform as a modulo-5 counter. The gating used here is a little complex, with a connection from the Q-bar output of FF3 to the J input of FF1, a connection from the Q output of FF3 back to its own K input, and an AND gate at the J input of FF3 whose inputs connect to Q1 and Q2. You might care to draw up your own truth table, to verify that this circuit does in fact have a counting modulo of 5.

It is not always possible to use the J and K inputs to achieve the gating required for a given modulo. With some modulos, the easiest way of making the counter "skip" the required number of states is to arrange for an AND gate to detect when the counter has produced the required maximum count, and then reset all the flipflops via their R inputs.

This approach can be particularly useful where it is desired to be able to "program" a counter to any one of a number of modulo factors. This is illustrated in Fig. 4. As you can see, the output of the 4-input NAND gate connects to the R inputs of the flipflops, and is therefore able to reset the counter whenever all four of its inputs are taken to the true (1) logic level simultaneously. Each of the four gate inputs may be connected to either the output of a FF, or to the 1 level; by setting the switches S1-4 to the appropriate combination the counter may thus be given virtually any modulo between 1 and 15.

With the switches set as shown, for example, the gate resets the counter whenever Q1, Q2 and Q4 are in the 1 state simultaneously. If you refer back to the truth table of Fig. 1, you will see that this combination first occurs immediately after the 11th input pulse. Hence the combination of switches shown effectively turns the circuit into a modulo-11 counter.

So far, we have considered counters which begin counting from zero, and count continuously upward until the maximum count is reached. These are often called "up counters" or incrementing counters. It is also possible to produce counters to do the opposite, jumping from zero to maximum count on the first input pulse, and then counting downwards. Not surprisingly these are called "down counters" or decrementing counters.

A simple ripple-carry down counter may be produced by connecting a chain of flipflops as in Fig. 5. Instead of connecting the clock inputs of the second and later flipflops to the Q outputs of the preceding FF, as before, they are now connected to the Q-bar outputs. This causes each FF to "carry-over" to the next whenever it sets, rather than when it resets.

By using gating, it is possible to produce a counter which may be programmed to count either up or down at will, by means of a control logic level. Not surprisingly, such counters are called "up-down counters".

While ripple-carry counters are adequate for many applications, there are

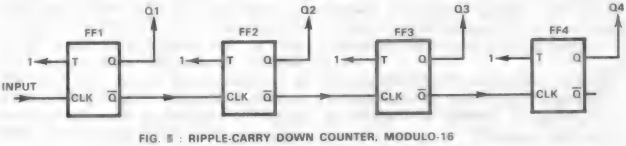


FIG. 5 : RIPPLE-CARRY DOWN COUNTER, MODULO-16

some situations where the progressive response of such a counter to the input pulses can cause problems. Mostly these arise because during the time taken for the changed input to "ripple down" the counter flipflop chain, the overall count present at the FF outputs may pass briefly through many spurious values.

Consider, for example, what happens when the counter of Fig. 1 has received 7 previous pulses, and then receives its 8th pulse. First, FF1 will reset, giving the count "0110" for a brief instant. Then FF2 will reset, changing the count briefly to "0010". Then FF3 will reset, giving a further brief count of "0000" before FF4 finally sets to give the final correct count of "0001", or binary 8.

As you can see, the counter does not change directly from binary 7 to binary 8, but briefly passes through binary 6, binary 4 and binary 0 before reaching binary 8. Although these "spurious" counting states may occur for only a small fraction of a micro-second, they may still be capable of causing trouble.

To cope with situations where a ripple-carry counter is not suitable, various types of "synchronous" counting circuits have been developed. As the name suggests, synchronous counters are designed so that all of their flipflop outputs change simultaneously, from one count to the next.

A simple modulo-16 synchronous counter is shown in Fig. 6, and you may care to compare it with Fig. 1. Note that here all of the FF clock inputs are fed with the input pulses, to ensure synchronous operation. The binary counting sequence is established by means of two AND gates, one a two-input gate feeding the T input of FF3 (G1), and the other a three-input gate feeding the T input of FF4 (G2).

The T input of FF1 is taken permanently to logical 1, so that as before this element toggles on every input pulse. However, the T input of FF2 is connected to Q1, so that FF2 is only able to toggle on every alternate input pulse. Gate G1 then ensures that FF3 is only able to toggle on every fourth input pulse, when both FF1 and FF2 are in the set state. Finally G2 ensures that FF4 is only able to toggle when FF1, FF2 and FF3 are all set, which occurs only once in every eight input pulses.

The circuit of Fig. 6 thus counts in normal binary fashion, with a truth table exactly the same as that shown for Fig. 1. The only difference is that the circuit of Fig. 6 changes cleanly and synchronously between each of the counting states.

Both ripple-carry and synchronous counters may be provided with parallel-load inputs, rather like the registers we looked at in the last chapter. This allows the counter to be preset, so that it effectively starts with a fixed count. A decrementing or down-counter provided with this facility may thus be used to count items in batches, by presetting it each time with the required batch size, having it count down with the individual items, and ring ■ bell or otherwise indicate when its count reaches zero.

The counters we have looked at this far have been based on binary counting. There are other types of counters encountered in digital circuits, some of which can offer advantages over binary counters in some situations.

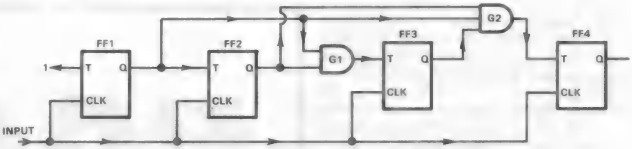


FIG. 6 : SYNCHRONOUS MODULO-16 COUNTER

The most well-known counters in this category are the "shift counters", so name because they are based on shift registers. There are two basic types of shift counter: the ring counter, and the twisted-ring or Johnson counter.

Essentially, a ring counter consists of a simple shift-right register whose serial output is looped back and connected to its serial input, to form a ring. A suitable bit pattern is then loaded into the register, which counts by shifting the pattern continuously around the ring.

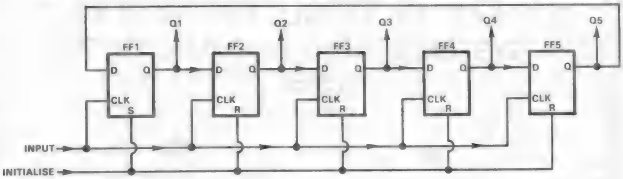


FIG. 7 : MODULO-5 RING COUNTER

INPUT PULSES	Q1	Q2	Q3	Q4	Q5
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5(0)	1	0	0	0	0

Usually the bit pattern loaded into the ring is either a single 1, or one less 1 than there are stages in the shift register—to give a single 0, in effect. Both these patterns allow the ring counter to have a counting modulo equal to the number of stages in the ring, whereas other patterns tend to produce a smaller modulo.

Note that there must be at least one 1 and one 0 in the pattern, for the ring counter to work at all. If the "pattern" were all 1's or all 0's (scarcely a pattern in the usual sense of the word!), there would be no way of telling its position in the ring—so the counter would not work.

A simple modulo-5 ring counter using D-type flipflops is shown in Fig. 7. Aş you can see, it is basically just a 5-stage shift register with Q5 tied back to the D input of FF1. However an "initialise" control line is provided, to set up the required bit pattern prior to counting. In this case the line connects to the S input of FF1 and the R inputs of the remaining stages, so that if the initialise line is taken briefly to logic 1, a single bit is loaded into the ring via FF1.

The truth table shows how the circuit counts, by passing the 1 around the ring.

Ring counters of this type may be produced with virtually any desired modulo, simply by using the same number of flipflops.

The Johnson or "twisted-ring" counter is like the normal ring counter in that it is formed from a simple shift-right register. However, in this case the serial input of the first FF is fed not from the last FF's Q output, but from its Q-bar output. This introduces ■ logical inversion or "twist" into the ring, because the first FF must always adopt the opposite state from that of the last.

INPUT PULSES	Q1	Q2	Q3	Q4	Q5
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	0
4	1	1	1	1	0
5	1	1	1	1	1
6	0	1	1	1	1
7	0	0	1	1	1
8	0	0	0	1	1
9	0	0	0	0	1
10(0)	0	0	0	0	0

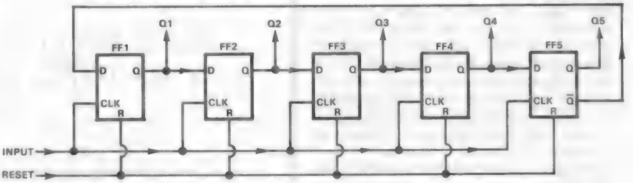


FIG. 8 : JOHNSON OR "TWISTED RING" COUNTER

The idea is shown in Fig. 8, which again shows a 5-stage counter. However, if you look at the truth table, you will see that this counter has a modulo of 10—twice the modulo of a normal 5-stage ring counter. This is probably the main feature of Johnson counters: the logical twist gives them a counting modulo equal to double the number of stages in the ring, and hence twice the modulo of a normal ring counter.

Note that like the normal ring counter, the Johnson counter must be initialised, or loaded with a suitable bit pattern before counting begins. The pattern may be any of the combinations which normally occur during its counting sequence, although in most cases it is easiest to make it either all-1's or all-0's. In Fig. 8 the latter is done, by tying all the flipflop R inputs together to form a reset line.

The individual flipflop elements in a Johnson counter are only called upon to toggle at a frequency equal to the input frequency of the counter divided by the modulo factor, as you can see from the truth table. Each flipflop only completes a full reset-set-reset cycle once in every 10 input pulses, for a five-stage Johnson counter.

While at first glance this may suggest that the Johnson counter configuration allows operation at higher frequencies, for ■ given type of flipflop device, this is not the case. Each flipflop must still make a 1-0 or 0-1 state transition in the time between consecutive input pulses, as shown by the truth table. This means that in practice the maximum operating frequency of a Johnson counter is the same as other counter configurations using the same flipflop elements.

Although the Johnson counter is twice as "efficient" as the normal ring counter, using half as many flipflops for a given modulo, there's ■ catch — it isn't able to count in an odd modulo. This is because its modulo is twice the number of stages, and 2N is always even regardless of N.

Chapter 11

Encoding & Decoding

Information is usually handled within digital systems in coded form, as this tends to be more economical of circuitry. At the interface between humans and systems, it therefore becomes necessary to perform encoding and decoding. This chapter looks at the basic ways in which these operations are performed.

As we have seen in earlier chapters, digital circuits and systems are often used to convey, process or acquire information. The information may consist of either numerical data or messages, or a mixture of both, but whatever its content it is normally capable of being expressed in human language using alphanumeric symbols.

While the information is normally in human language form when it ultimately emerges from a digital system, and also before it may be fed into a system, within such a system it generally exists as an electrical, magnetic or perhaps even an optical representation. And typically this representation consists of suitable combinations of two discrete values of voltage, current, or magnetic or optical flux.

Although the representation may be a direct one—for example, using 26 different digital signals to represent the 26 alphabetic characters, this is rather inefficient and is not usually done. Instead the information is usually handled in encoded form, as we have seen, using each of the various unique truth-value combinations of a group of binary digits or "bits" to represent one of the usual human language symbols.

Encoding gives a considerable increase in information-carrying efficiency, because the truth-value combinations provided by a group of bits rises exponentially with group size. If there are N bits in the group, the number of unique truth-value combinations available for encoding is equal to the Nth power of 2.

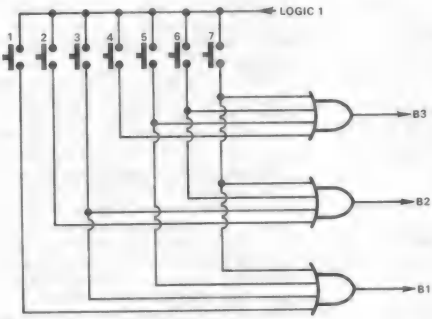
Four bits accordingly provide 16 truth-value combinations, enough to encode the 10 decimal digit symbols with 6 combinations to spare. And as we saw in chapter 7, seven bits provide no less than 128 truth-value combinations, sufficient to encode all of the alphanumeric and control characters of the ASCII code.

When information is to be fed into a digital circuit or system then, it is generally first processed by a type of circuit known as an **encoder**. Conversely when information is to emerge from a digital circuit or system, it is generally processed by a complementary type of circuit known as a **decoder**. In the remainder of this chapter we will look at the way in which these encoder and decoder circuits operate.

As the name itself implies, the function of an encoder circuit is to accept any one of a number of different input signals, and generate in response to each one a code "word" or group of bits having a preassigned unique combination of truth values. As we have seen, the number of bits in the generated code word will depend upon the number of different input signals the encoder is required to handle.

In its most basic form, an encoder consists of a set of multi-input OR gates, with one gate for each of the code bits.

A simple 3-bit encoder is shown in Fig. 1. Here there are seven inputs to the encoder, provided by a set of pushbutton switches, and the encoder is designed to provide output code words which are simply the binary equivalents of the digits 1-7. This is shown in the truth table.



INPUT	B1	B2	B3
(0)	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

FIG. 1 : BASIC 3-BIT BINARY (OCTAL) ENCODER

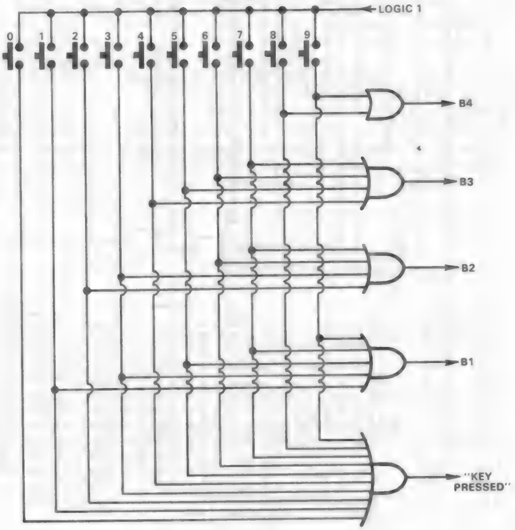


FIG. 2 : 4-BIT 8421 BCD ENCODER WITH "KEY PRESSED" OUTPUT

The actual encoding is done by connecting each input line to an input of those gates required to produce a logic 1 output for the appropriate output code. Thus the "1" input line connects only to an input of the B1 gate, while the "3" input connects to input of both the B2 and B1 gates. Similarly the "7" input connects to inputs of all three gates.

Note that when none of the input buttons is pressed, all three gates will be disabled, and the output lines will all be at the 0 logic level. This corresponds to the eighth possible truth-value combination provided by three bits, and is therefore shown in the truth table for completeness. However with this simple circuit it would not be possible to have a "0" input button, because it would not be connected to the gates and would therefore be ineffective.

If all of the possible truth-value combinations of an encoder are to be utilised for coding, it is necessary to modify the basic encoder circuit by adding a further multi-input OR gate, with one input connected to all input lines. This then produces a logic 1 output whenever any of the keys are pressed, allowing even the "all zeros" truth-value combination to be used for coding.

This is shown in Fig. 2, where as you can see the 10-input gate at the bottom of the diagram is used to generate the required "key pressed" signal. This can be used by subsequent circuitry to trigger processing of the generated code word.

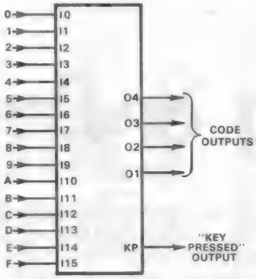


FIG. 3 : INTEGRATED 4-BIT BINARY (HEXADECIMAL) ENCODER

INPUT	OUTPUTS				
	O1	O2	O3	O4	KP
(NONE)	0	0	0	0	0
0	0	0	0	0	1
1	1	0	0	0	1
2	0	1	0	0	1
3	1	1	0	0	1
4	0	0	1	0	1
5	1	0	1	0	1
6	0	1	1	0	1
7	1	1	1	0	1
8	0	0	0	1	1
9	1	0	0	1	1
10(A)	0	1	0	1	1
11(B)	1	1	0	1	1
12(C)	0	0	1	1	1
13(D)	1	0	1	1	1
14(E)	0	1	1	1	1
15(F)	1	1	1	1	1

The encoder shown in Fig. 2 generates four code bits, and is arranged to perform encoding of 10 inputs into the 8421 BCD code. You may care to draw up a truth table based on the circuit connections, to verify that this is the case.

Although encoder circuits may be produced by connecting up discrete gates along the lines of Figs. 1 and 2, integrated circuit manufacturers have largely made this unnecessary by producing prepackaged encoder circuits. These may be treated as functional "black boxes", with a set of inputs and output whose relationships are given by a truth table.

This is illustrated in Fig. 3, which shows the logic diagram and truth table for an integrated 4-bit binary encoder which provides encoding for all 16 possible inputs, in addition to a "key pressed" or KP output. Such an encoder could be used to handle numerical information in either hexadecimal form, as shown, or in normal decimal form by simply ignoring inputs I10-I15.

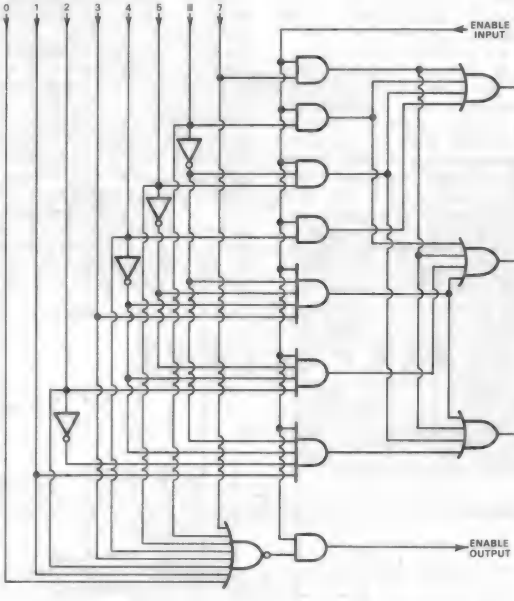
Simple encoders of this type are typically used to accept information from small input keyboards, and prepare equivalent code words for use in a computer, calculator or other digital system.

Where a large number of input keys are to be encoded, in theory it is possible to expand the basic OR-gate encoding scheme to provide the required number of code bits and input lines. However, in practice this can become rather clumsy, involving numerous connections and gates with large numbers of inputs.

As a result designers of large keyboard encoders tend to use one of a number of modified encoding schemes. Mostly such schemes involve connection of the keys into a two-dimensional array or "matrix", and augmenting the basic OR gate system with some sort of sequential scanning. This will be discussed in chapter 12.

For the present, there is one further type of simple encoder circuit which should be discussed. This is the so-called **priority encoder**.

A priority encoder is not generally used



INPUTS									OUTPUTS			
EN	0	1	2	3	4	5	6	7	B1	B2	B3	EN
0	X	X	X	X	X	X	X	X	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0	0
1	X	1	0	0	0	0	0	0	1	0	0	0
1	X	X	1	0	0	0	0	0	0	1	0	0
1	X	X	X	1	0	0	0	0	1	1	0	0
1	X	X	X	X	1	0	0	0	0	0	1	0
1	X	X	X	X	X	1	0	0	1	0	1	0
1	X	X	X	X	X	X	1	0	0	1	1	0
1	X	X	X	X	X	X	X	1	1	1	1	0
X = "DON'T CARE" (EITHER 1 OR 0)												

FIG. 4 : 3-BIT PRIORITY ENCODER

for encoding information from a keyboard, but for responding to logic signals generated by preceding digital circuits. Like the simple encoder of Figs. 1, 2 and 3 it produces a unique output code word for each input, but there is an important difference. The various inputs are arranged in order of "priority", so that if two or more input signals are applied at the same time, the encoder generates the output code corresponding to the input with the highest priority, effectively ignoring the others.

Why do this? Well, there are quite important applications in digital circuitry where a system—such as a computer or controller—must respond to a number of signals, but some of the signals must be regarded as more "important" or more "urgent" than others.

An example of this is the "interrupt" facility on many computers. Here the control circuitry of the computer is provided with a means whereby it can be forced to "drop what it is doing", and handle an urgent data input or other transaction. There are often a number of interrupt sources, and it is generally necessary to give them an order of priority—so that the most important transaction cannot be interrupted by less important ones, and so on. A priority encoder is generally used at the interrupt inputs, to both provide an encoded identification for the interrupt signal source, and to ensure that the source with the highest priority takes precedence.

A three-bit priority encoder is shown in Fig. 4. As you can see, it has a set of AND gates ahead of the actual encoding OR gates, to establish the priority order of the signal inputs.

If you trace through the logic, you will find that input 7 takes priority over all of the other signal inputs. Then if input 7 is at logic 0, input 6 takes priority over the remaining inputs, and so on.

The encoder also has an enable input and output, to allow a number of similar circuits to be cascaded if a larger priority encoder is required. The encoder only works if the enable input is at logic 1, and only produces a logic 1 level at its own enable output if (a) the enable input is at logic 1, and (b) none of its own inputs are at logic 1.

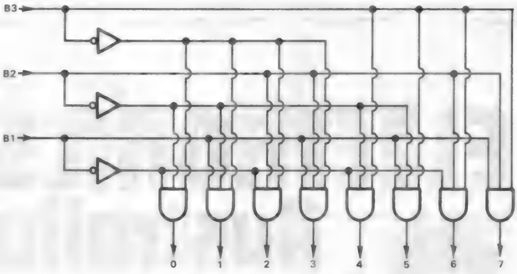


FIG. 5 : BASIC 3-BIT BINARY (OCTAL) DECODER

The enable input would normally be driven by the enable output of a "higher priority" encoder, while the enable output could be used to drive the enable input of a "lower priority" encoder. The enable input would otherwise be connected to logic 1, if the encoder were being used alone or was the one in a group with the highest priority. Similarly the enable output could be ignored if the encoder were used alone, or was the one in a group with the lowest priority. Alternatively it could be used as an active-low "signal present" output, equivalent to the "key pressed" output of Figs 2 and 3.

Having looked briefly at basic encoder circuits, let us now turn to the circuits which perform the complementary logic function: decoding.

Essentially, a decoder circuit is one which accepts information in encoded form, and regenerates signals corresponding to the original information before it was encoded.

Practical decoders do this by having an array of AND gates, each of which is arranged to respond to a particular unique combination of truth-values on the incoming code bit lines. Normally there is a gate provided to respond to every possible truth-value combination of the code bits, so that all combinations may be decoded.

A basic 3-bit binary encoder is shown in Fig. 5. As you can see, it has eight AND gates, each arranged to respond to one of the eight truth-value combinations possible with 3 bits. Apart from the AND gates there are three inverters, used to provide complementary versions of the incoming code signals to those gates which need them.

Thus to respond to the code combination 101, corresponding to octal 5, the appropriate AND gate must be fed with B1, B3 and the complement of B2. Similarly to respond to 011, corresponding to octal 6, the next AND gate must be fed with B2, B3 and the complement of B1.

Note that only one output of the decoder can be at logic 1 level at any one time, because each gate responds to a unique truth-value combination of the code bits, and there is a gate for all possible combinations.

The simple binary decoding system of Fig. 5 can easily be expanded, by adding further gates. Fig. 6 shows a 4-bit binary decoder, which as you can see is similar to the 3-bit decoder except that the decoding gates here have 4 inputs each, and four inverters are required. This decoder could be used for decoding hexadecimal information, or for decoding information in one of the BCD codes if outputs A-F were ignored.

It may seem wasteful to suggest that one would use a decoder like that of Fig. 6 to decode BCD information, by ignoring 6 of the outputs, but the fact is that nowadays this is very often the cheapest way of doing the job—thanks to modern IC technology. Full 4-bit binary encoders

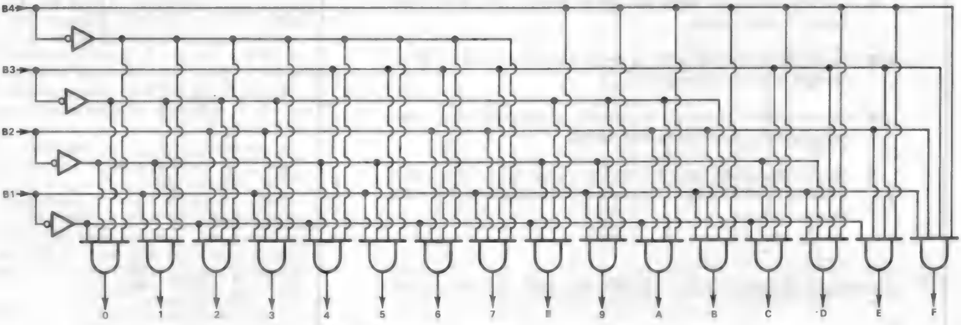


FIG. 6 : 4-BIT BINARY (HEXADECIMAL) DECODER

are available as single integrated circuits, at a price considerably lower than if a 10-output BCD decoder were built up from separate gates.

Of course before integrated decoders became available, designers had to produce their own decoders using separate gates, and there was a strong incentive to be as economical as possible. This being the case, they found all sorts of ways of reducing the number of gates required—particularly where only BCD decoding was required.

It was soon realised, for example, that if a decoder was only going to be used to decode BCD information it didn't really need to sense every code bit for the decoding of each and every truth-value combination. With some combinations, it became sufficient to sense only some of the bits, because these were adequate to distinguish that combination from the others.

This concept of "sufficient definition" was used to produce minimal logic BCD decoders, like that shown in Fig. 7. This is a decoder for 8421 BCD code, and as you can see it is somewhat simpler than if full decoding were used for each of the 10 outputs. Only the "0" and "1" outputs are fully decoded, while the "8" and "9" outputs have very simple decoding using 2-input gates. The remaining outputs use 3-input gates, which are still simpler than the 4-input gates which would be used for full decoding.

Yet if you trace through the logic, you will find that this decoder is quite adequate to decode 8421 BCD code. Each of the ten truth-value combinations which are "legal" in this code will produce a single and corresponding output.

Of course the problem with this sort of decoder is that if a code word becomes changed somehow into one of the truth-value combinations which are "illegal" in the code being used, the decoder will not handle it correctly. Most likely two outputs will respond at once, which can cause all sorts of trouble in following circuitry.

Because of this problem, minimal logic decoders like that shown in Fig. 7 are not often used nowadays. There are 10-output decoders made for BCD encoding, but these generally use full decoding for all of the outputs to prevent decoding errors. In effect they are simply truncated versions of the full 4-bit binary decoder of Fig. 6, with only 10 outputs in place of the full 16.

Incidentally although the decoders in Figs. 5, 6 and 7 are all shown having "true" outputs, which go to logic 1 when selected, this is not essential. Many practical decoders are made to have "false" or "active-low" outputs, which go to the logic 0 level when selected. This is done simply by using NAND gates for the decoding, instead of AND gates.

Decoders of both types are made and used, as each has advantages in certain applications. It depends largely upon the

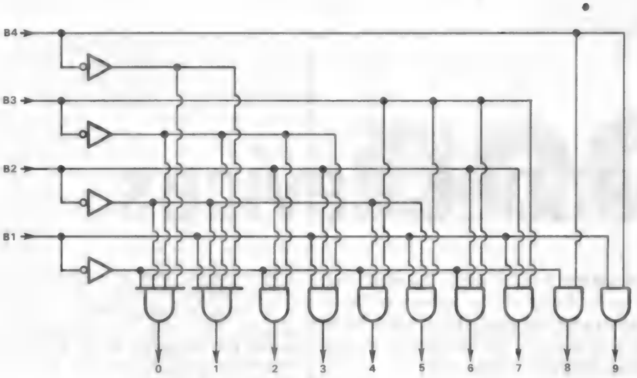


FIG. 7 : MINIMAL-LOGIC 8421 BCD DECODER

circuitry or devices which are to accept the decoded information — whether it can most easily cope with the information in "active high" or "active low" form.

Before we leave the subject of decoders for the present, there is a further type of circuit which should perhaps be described. This is not really a decoder at all, although the circuits concerned are usually described as such for convenience. There are only two common varieties, described respectively as the "BCD to 7-segment decoder" and the "hexadecimal to 7-segment decoder".

As the names suggest, these circuits are used to accept either BCD or hexadecimal information encoded in 4-bit form, and produce signals capable of driving one of the various types of. 7-segment display device. We will look at display devices in the next chapter, and it would not be appropriate to describe 7-segment displays here in detail. For the moment it should be sufficient to visualise them as having seven display segments arranged in a "squared-8" pattern, so that by activating the appropriate segments a variety of symbols may be displayed.

Fig. 8 shows how a "BCD to 7-segment" decoder is used to drive such a 7-segment display device from incoming information in 8421 BCD code. As you can see from the truth table, the "decoder" takes each incoming BCD code combination and produces signals on the seven outputs a-g, to produce the appropriate display. For example when the input code is 1010, the circuit produces a logic 1 at outputs a,c,d,f and g, to display a 5.

Of course what the circuit of Fig. 8 actually performs is not decoding, but code translation—it takes information in 4-bit BCD code and translates it into the special 7-bit code required to drive the display. The final decoding of the information into human-readable form takes place in the 7-segment display itself.

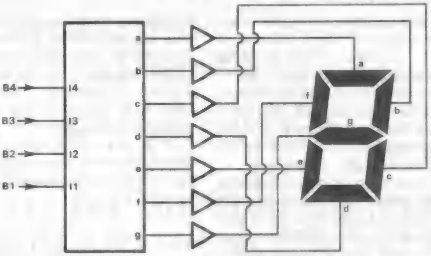
Similarly a "hexadecimal to 7-segment decoder" takes incoming 4-bit binary code, and generates the appropriate combinations of output truth values to produce the 16 appropriate hexadecimal display symbols—in slightly modified form. Apart from the familiar decimal symbols 0-9, the additional hexadecimal symbols are displayed as "A", "b", "C", "d" and "E"—the two lower-case symbols being used to avoid confusion with the numerals 8 and 0.

In short, then, a "hexadecimal to 7-segment decoder" is again not really a decoder at all, but a code translator.

Both 7-segment translator circuits tend to be grouped in with true decoder circuits because they are used in similar applications.

Finally, it should be noted that in addition to providing basic decoding circuits of the type shown in Figs. 5, 6 and 7, some manufacturers also provide integrated circuits which combine a decoding circuit with a set of flipflop latches, capable of performing storage.

Such "latch-decoders" may have the storage latches either before or after the decoding circuitry, although it is more common for the latches to be on the input side as this requires fewer flipflops. But both types of latch-decoder have their uses, as many decoding applications also call for the decoded information to be stored for some finite time.



INPUT CODE				DECIMAL	SEGMENTS DRIVEN						
B1	B2	B3	B4		a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
0	1	0	0	2	1	1	0	1	1	0	1
1	1	0	0	3	1	1	1	1	0	0	1
0	0	1	0	4	0	1	1	0	0	1	1
1	0	1	0	5	1	0	1	1	0	1	1
0	1	1	0	6	0	0	1	1	1	1	1
1	1	1	0	7	1	1	1	0	0	0	0
0	0	0	1	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	0	0	1	1

FIG. 8 : INTEGRATED 8421 BCD TO 7-SEGMENT DECODER—
WITH DRIVERS AND DISPLAY

An example is in frequency counters, where the BCD output of each decade counter is normally fed via a latch-decoder to the display device. The storage provided by the latches allows each count to be displayed while the next count is taking place, so that each reading is stable and readable until it is replaced by the next.

In early counters which used a plain decoder without latches, each reading only lasted a short time before the display "rolled again", as the counter began the next counting. This made the early instruments quite difficult to read at times, compared with modern instruments having latch-decoders.

Chapter 12

Basic readout devices

The output information from digital systems must often be displayed visually in numerical form. This chapter looks at the variety of basic readout devices used for output displays of this type. Those described are gas-discharge tubes, light-emitting diodes, fluorescent displays, incandescent filament tubes and liquid-crystal panels.

In many applications of digital electronics, the output from a system is interpreted as a number which must be displayed directly to the operator in visual form. A good example of this is provided by digital instruments such as frequency counters and digital voltmeters (DVMs), where the number displayed corresponds directly to Hertz or volts, respectively.

Over the years many different types of display device have been used to provide this sort of readout, some with more success than others. To cover all of them here would take a great deal of space, and in many cases the effort spent would be largely wasted as the devices concerned have long since faded into obscurity. We will therefore look only at the devices and techniques which have survived the test of time, and are still in current use.

Probably the first really successful digital display device to be developed was the shaped-cathode gas discharge tube, shown in Fig. 1. These are often given the generic name of "Nixie" tubes, although this name is strictly a trademark of the Burroughs Corporation in the USA who first developed tubes of this type.

Basically these tubes are a development from the simple neon lamp, which has two metal electrodes sealed inside a glass tube containing neon gas and a trace of mercury vapour. When a potential of about 75 volts DC is applied between the electrodes, the gas inside the tube ionises and a glow discharge is produced in the immediate vicinity of the cathode (more negative electrode).

It is this confinement of the glow discharge to the immediate vicinity of the cathode which is exploited in the shaped-cathode display tube. Here there is not one cathode but ten, and each is shaped in the form of one of the ten decimal numerals 0-9. The ten cathodes are stacked one behind the other, as shown, with the anode in the form of a fine wire mesh in the front. The tube may thus be arranged to display any of the numerals at will, simply by activating the appropriate cathode.

Typically this is done using a circuit of the type shown. The anode of the tube is connected to a source of around 150V DC, via a resistor R. Each of the cathodes is connected to ground (negative) via a high-voltage switching transistor, with the transistors controlled by the outputs of a BCD-to-decimal decoder. The decoder thus causes the tube to display the decimal numeral corresponding to the BCD code presented to its input.

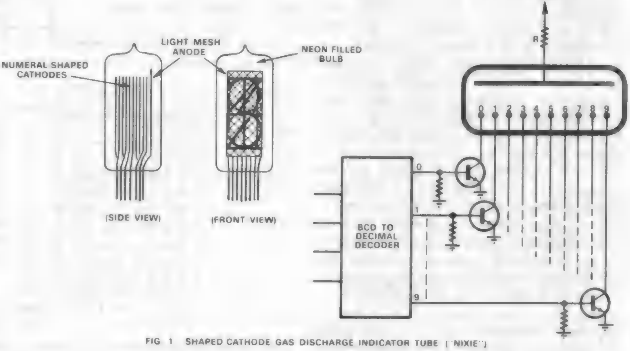


FIG. 1 SHAPED CATHODE GAS DISCHARGE INDICATOR TUBE ("NIXIE")

The anode series resistor R is required because although the tube has a voltage drop of around 70V when ionised, it requires at least 120V for ionisation to take place. The resistor absorbs the difference in voltage, and also tends to maintain the tube current relatively constant despite differences between the various cathodes. Typically the resistor value is between 50k and 100k.

While the shaped-cathode display was used in a great many digital readouts, it had one main disadvantage. This was that the discharge produced by the cathodes near the rear of the tube tended to be obscured by the cathodes in front, making them difficult to read.

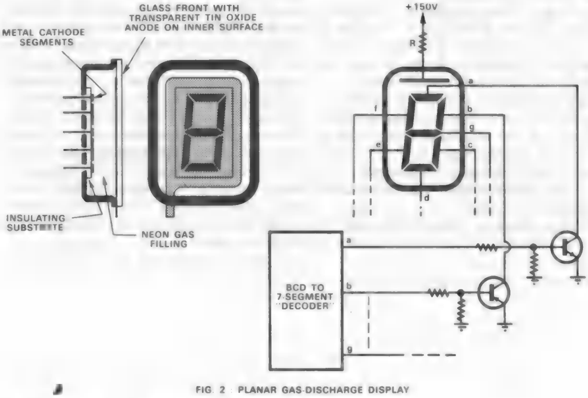

GLASS FRONT WITH TRANSPARENT TIN OXIDE ANODE ON INNER SURFACE
METAL CATHODE SEGMENTS
INSULATING SUBSTRATE
NEON GAS FILLING
+150V
R
a
b
c
d
e
f
g
BCD TO 7-SEGMENT "DECODER"
a
b
g

FIG. 2 PLANAR GAS-DISCHARGE DISPLAY

In order to obviate this problem, designers came up with the planar type of gas discharge display shown in Fig. 2. This uses ■ somewhat flatter construction, with the multiple cathodes all in the same plane. But in this case there are only seven cathodes, arranged in the now-familiar squared-8 or "seven segment" format, so that the various decimal numerals may be displayed by activating combinations of segments simultaneously.

Thus a "1" is displayed by activating cathodes b and c, for example, or a "6" by activating cathodes c, d, e, f and g. The cathodes are again controlled by high-voltage switching transistors, as shown, but in this case the transistors are driven by ■ BCD-to-7-segment code translator rather than a decoder.

Note that the anode of a planar gas-discharge display is not usually a wire mesh, but rather a transparent electrode of tin oxide deposited on the inside of the glass front plate. This tends to give improved visibility also, over and above that provided by the single-plane type of display.

Of course the planar gas discharge display shares with the original shaped-cathode type the need for ■ high-voltage power supply, and for switching transistors capable of withstanding up to 120V or so. This makes both types rather unsuitable for use in low voltage systems.

Because of this problem gas discharge devices have tended to lose favour in recent years, with designers opting more for display devices capable of operating from lower voltage supplies.

The most popular of these is the light-emitting diode or LED display, shown in Fig. 3. Like the planar gas-discharge display this uses the 7-segment display format, but here the sources of light for the seven segments are individual LED chips. The chips are mounted on a common substrate, in positions which correspond to the centre of each display segment.

The individual chips are typically quite small—less than 1mm square. This makes direct viewing of the chips practical only for very small displays, such as those on digital watches and pocket calculators. Even for these applications, moulded plastic lenses are usually fitted in front of the chips to provide optical magnification.

For larger displays, the technique shown in Fig. 3 is generally used. Here a moulded plastic block which mounts in front of the chips on the substrate is provided with tapered slots, the inside of which are metallised to form mirrored "light pipes". In front of these again is a red tinted diffusing filter, and together the mirror pipes and filter effectively spread the light from the LED chips out to form uniformly lit segments.

Electrically the LED chips are usually connected either with their cathodes commoned, as shown, or with their anodes commoned. Devices with both arrangements are made and used, as each has advantages in certain applications. The advantages of having one side or the other of the chips commoned are twofold: the number of device connections is reduced, lowering package cost, and the common connection allows convenient electrical control of the display digit as a whole.

LED displays tend to have high reliability and a long operational life, as a result of the solid state construction. And conducting LED chips have a voltage drop of around 1.7V, making them quite compatible with most digital system power supplies. These features have made them very popular as digital displays in ■ wide variety of applications, although as a LED chip typically requires an average current drain of around 15-20mA for ■ reasonably bright output, a full 7-segment digit display tends to have a current consumption of around 140 milliamps.

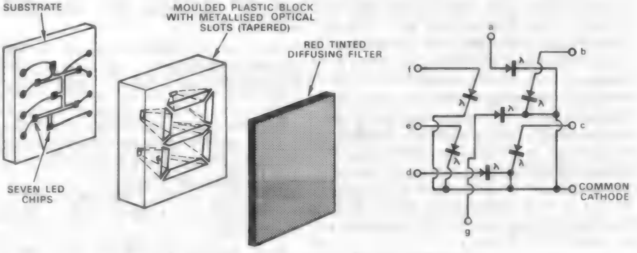

SUBSTRATE
MOULDED PLASTIC BLOCK WITH METALLISED OPTICAL SLOTS (TAPERED)
RED TINTED DIFFUSING FILTER
SEVEN LED CHIPS
a
b
f
e
c
d
COMMON CATHODE
g

FIG. 3 : SEVEN-SEGMENT LED DISPLAY (COMMON CATHODE)

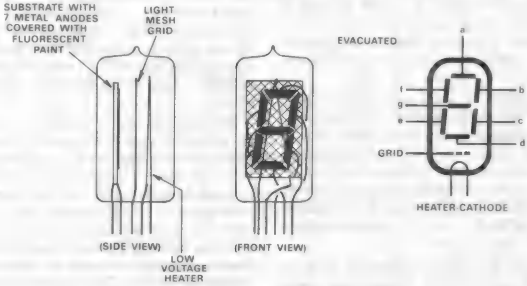


FIG. ■ : FLUORESCENT 7-SEGMENT DISPLAY TUBE

Because of the current drain of LED segments, this type of display is not easily driven by MOS logic devices. Designers of systems employing MOS logic have therefore tended to use alternative display devices, which involve lower operating current levels—or at least lower control currents.

One such device which has been popular particularly among Japanese designers is the fluorescent display tube, shown in Fig. 4. This is essentially ■ modern adaptation of the early "magic eye" tuning indicator tubes, used in valve radio sets in the 1930's and 1940's.

The device is a thermionic vacuum tube, with an electrically heated fine wire filament in the front. The filament is typically connected to a supply of around 1V to 1.5V AC or DC, and draws around 40mA. This heats it to a point just short of incandescence, sufficient to release free electrons from the surface of the wire, but not sufficient to make the wire visible.

At the rear of the tube is a substrate with seven metal anodes, shaped and laid out in the familiar 7-segment format. The anodes are covered with fluorescent paint, which glows with a greenish-blue light when struck by electrons. Any anode segment may thus be caused to glow by making it positive with respect to the heater filament, so that some of the free electrons released by the filament are attracted to the anode and made to impinge upon the fluorescent paint coating.

The positive anode voltage required to produce this action need only be around 20-25V, which is well within the drive capability of MOS logic circuits. The anode current levels are quite low, in the order of tens to hundreds of microamps.

Many fluorescent displays are provided with a mesh grid electrode between the heated filament and the anodes, as shown in Fig.4. The grid is used to control the display as a whole, where it must be switched on and off rapidly. This is done using a negative bias on the grid, which repels the electrons and prevents them reaching the anodes.

All of the display devices we have looked at this far tend to have relatively modest light output. While they are generally quite adequate in most applications, they thus share ■ common problem: a tendency to "wash out" and become hard to read in conditions of high ambient lighting levels.

A display device which is somewhat

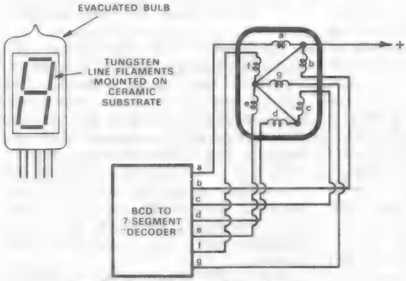

EVACUATED BULB
TUNGSTEN LINE FILAMENTS MOUNTED ON CERAMIC SUBSTRATE
a
b
c
d
e
f
g
+
BCD TO 7-SEGMENT "DECODER"
a
b
c
d
e
f
g

FIG. 5 FILAMENT 7-SEGMENT DISPLAY

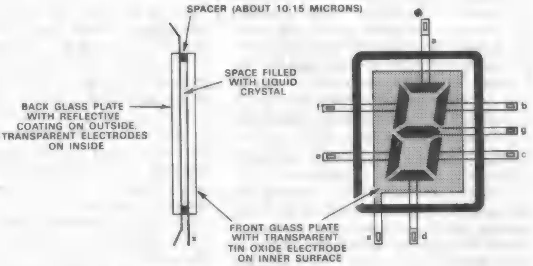

SPACER (ABOUT 10-15 MICRONS)
SPACE FILLED WITH LIQUID CRYSTAL
BACK GLASS PLATE WITH REFLECTIVE COATING ON OUTSIDE. TRANSPARENT ELECTRODES ON INSIDE
FRONT GLASS PLATE WITH TRANSPARENT TIN OXIDE ELECTRODE ON INNER SURFACE
x
a
b
f
g
e
c
d

FIG. 6 LIQUID-CRYSTAL DISPLAY

better in this respect is the directly-viewed filament display, shown in Fig. 5. This is basically a development of the conventional incandescent lamp, with seven tungsten wire filaments arranged in the standard 7-segment format. However, the filaments are here operated at a bright red heat—around 1400 degrees K (Kelvin)—compared with the white-hot 2500 degrees K temperature used with normal household lamps. This gives a very significant improvement in reliability, together with an average life expectancy of over 100,000 hours.

And despite the apparent fragility of the fine filaments, the displays turn out to be surprisingly rugged. So much so that in view of their high visibility in bright conditions they are widely used in high-shock applications such as aircraft cockpit displays

The ability to provide a highly visible display in conditions of high ambience is also possessed by the most recent type of readout device, the liquid-crystal panel. This competes with ambient lighting not by attempting to out-shine it, but by using it, ie, the liquid-crystal panel is basically an optical filter or modifier of existing light rather than a source.

Actually there are a number of different types of liquid-crystal display rather than a single type. All depend upon the rather unique properties of liquid-crystal materials, and the way these properties can change when the materials are subjected to an electric field. But different displays make use of different properties, or put the same property to different use.

One type of display make use of the property whereby a liquid crystal may be optically transparent when its molecules are at rest, but becomes cloudy due to molecular turbulence when an electric field is applied.

This type of display is known as the "dynamic-scattering" type of liquid-crystal panel, and is the type most widely used.

The other main type makes use of the fact that some liquid-crystal materials may be made to exhibit a light-polarising ability, which is again affected by an applied electric field. This type of liquid-crystal display is known as the "field-effect" type.

Both dynamic-scattering and field-effect displays may be made in either of two forms: transmissive and reflective. With transmissive displays the panel acts as a selective filter, modifying light which passes through it from behind. Reflective displays act instead as selective mirrors, modifying light incident on them from the front.

The construction of a dynamic-scattering reflective type of display is shown in Fig. 6. As you can see it is basically two sheets of glass sandwiching a very thin layer of liquid crystal. The inside of the front glass sheet is covered by a transparent electrode, usually of tin oxide, while the inside of the rear sheet has seven similar electrodes laid out in the familiar 7-segment format. The outside of the rear glass sheet also has a reflective mirror coating.

With the rear segment electrodes at the same potential as the front electrode, the molecules of the liquid crystal are at rest and the display appears bright due to the clear optical path to and from the rear mirror. However, if an external voltage is applied between any of the rear electrodes and the front electrode, the liquid crystal molecules immediately in front of the active segments become turbid, clouding the optical path. This makes the desired display evident, as a dark digit displayed against a light background.

Liquid-crystal displays are the most efficient of all in terms of electrical power consumption—typical power consumption is in the order of microwatts. They are also relatively low in cost, and these two features make them attractive for battery-operated consumer applications such as pocket calculators and digital watches.

It seems likely that liquid-crystal displays will be used much more widely in the future, although there are still some problems to be solved. One is response speed, as at present they are very much slower than other types of display in terms of turn-on and turn-off times. Another problem is display appeal—at present liquid-crystal displays tend to look drab and grey, compared with the other types. Until recently they also had a rather limited operational life, although this problem now appears to have been largely solved.

Before closing this introductory look at digital display devices, there is one point which should be noted. Although all of the examples shown in the diagrams have been single-digit displays, many of the different types of display device are also made in multi-digit form—providing 2, 3, 4, 5 or more digit displays in the one package.

The actual operation of such multi-digit displays is exactly the same as for single-digit displays, although the method of driving them from the associated logic often differs. A technique often used with multi-digit displays is multiplexing, which both reduces the number of connections required for the displays themselves, and also simplifies the driving circuitry.

We will look at multiplexing and demultiplexing techniques in the next chapter.

Chapter 13

Multiplexing

The technique of multiplexing is frequently used in digital electronics to minimise the number of circuits or communication channels required to handle multiple signals. In this chapter the author looks at the technique of multiplexing, at multiplexer and demultiplexer circuits, and at important applications of the technique.

Frequently a number of digital signals must be processed identically, or sent together over a significant distance. Where this must be done, it is generally desirable to minimise the number of processing circuits involved, or the number of cables or data channels required, in order to reduce the overall system costs and improve reliability.

One common and important way of doing this is multiplexing, which we will look at in this chapter. The word "multiplex" comes from two Latin words, "multi" meaning many, and "plecto" meaning to braid or interweave together. So literally, multiplexing is a technique where a number of signals may be interwoven so that they may be passed through a relatively small number of circuits or channels.

Strictly speaking there are two different types of multiplexing, one known as time-division multiplexing and the other as frequency division multiplexing. However the latter is used almost solely in communications, and is largely an analog technique. It is time-division multiplexing which is used much more widely in digital electronics, and this is the type of multiplexing which we will look at here.

The basic principle of time-division multiplexing is quite simple, as shown in Fig. 1. Two commutating switches are used, one at the input of the processing circuit or communications channel, and the other at the output. The rotors of the switches are rotated continuously at the same speed, and in phase synchronism.

The input switch cyclically samples the various input signals, so that each input signal is connected to the circuit or channel for a proportion of each revolution. Here there are six inputs and a six-segment commutating switch, so each signal is connected to the circuit or channel for one-sixth of the time.

The signal actually fed through the circuit or communications channel is thus not continuous, but an interleaved series of cyclic samples of the individual inputs.

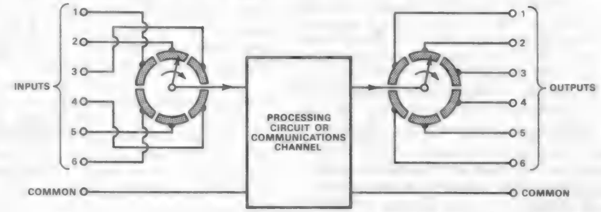


FIG. 1 : BASIC PRINCIPLE OF MULTIPLEXING

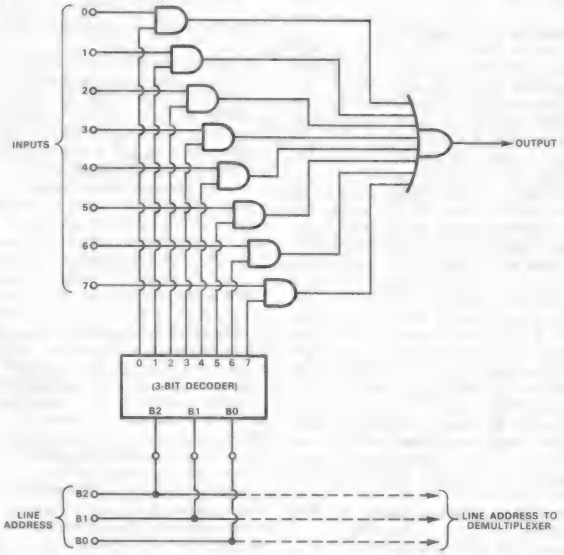


FIG. 2 : 8-LINE TO 1-LINE MULTIPLEXER

At the output of the circuit or channel, the input multiplexing process is reversed by the second commutating switch, which acts as a distributor or demultiplexer. As the output corresponding to each input signal sample is produced by the circuit or channel, it is routed to its corresponding output line by the switch rotor.

Note that in order to convey signal changes faithfully the multiplex sampling must be done either at a much higher rate than signal changes, or synchronously with them.

In digital logic circuitry, the equivalent of the input sampling switch of Fig. 1 is the multiplexer circuit. An 8-line to 1-line multiplexer circuit is shown in Fig. 2, and as you can see it consists of a set of eight AND gates together with an 8-input OR gate and a 3-bit decoder.

Each AND gate controls one of the input lines, while the outputs from the AND gates are effectively combined by the OR gate to produce a common output line. The AND gates are enabled by the eight output lines of the 3-bit decoder, and as only one output of the decoder can be at the logic 1 level at any one time, only one AND gate can be enabled at any one time.

Which of the inputs of the multiplexer is connected to the output may therefore

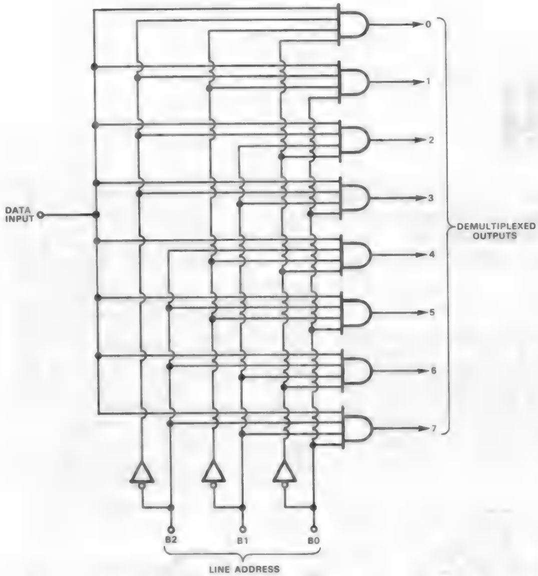


FIG. 3 : 1-LINE TO 8-LINE DEMULTIPLEXER

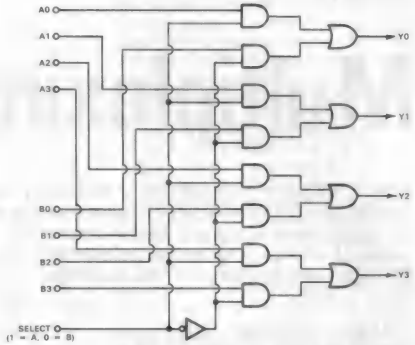


FIG. 4 : QUAD 2-LINE TO 1-LINE MULTIPLEXER OR "DATA SELECTOR"

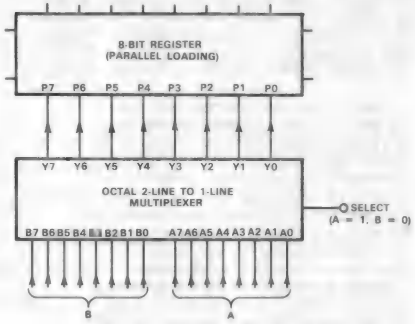


FIG. 5 : USE OF A MULTIPLEXER FOR SELECTION OF REGISTER INPUT DATA

be controlled merely by changing the 3-bit binary code applied to the decoder inputs, B0, B1 and B2. The binary code applied to these inputs thus becomes a "line address". A line address of 000 causes input line 0 to be connected to the multiplexer output, while a line address of 101 causes input line 5 to be connected instead, and so on.

To produce cyclic sampling of the input lines for multiplexing, all that is necessary is to derive the line address codes from a 3-bit binary counter which is driven in turn by a clock pulse generator. The counter will then produce a continuous sequence of binary code numbers, causing the multiplexer to scan the eight input lines in cyclic order.

Ideally the counter used to produce the line addresses should be a synchronous type, to avoid "glitches" (see chapter 10).

Note that the line address code used to drive the multiplexer is normally also used to drive the demultiplexer at the other end of the system or communications channel, as indicated in Fig. 2. This is to ensure that the two ends remain in synchronism.

The digital logic equivalent of the output distributor switch of Fig. 1. is the demultiplexer circuit. A 1-line to 8-line demultiplexer circuit which complements the multiplexer of Fig. 2 is shown in Fig. 3. As you can see, it is actually quite similar to a 3-bit binary decoder (compare it with Fig. 5 of chapter 11). The only difference is that each of the AND gates has an additional input, and the additional inputs are all connected together to form the common data input line.

As with a decoder, only one of the AND gates can be enabled at any one time, and the 3-bit line address again determines the enabled gate. Hence when the line address is 000, any signal appearing at the data input is routed automatically to output 0; when the line address is 011, it is routed to output 3; and so on.

Hence by driving the line address inputs of the demultiplexer from the same counter used to drive the multiplexer circuit, the multiplexed data is automatically "unravelled" back into its separate parts.

Note that a multiplexed data communications channel using the circuits of Figs. 2 and 3 would require a total of four lines connecting input and output: the single line carrying the multiplexed data, together with three lines to carry the line address information. However, the system would be capable of handling eight signals, twice the number normally carried by four lines. (Note that as in most logic circuits, all of the lines use a common "earth" return.)

We could double this capacity by using a 16-line to 1-line multiplexer circuit, together with the corresponding 1-line to 16-line demultiplexer. Yet this would involve only a single additional line, to carry the fourth address bit—a total of only five lines.

In general terms, multiplexing allows "N" communications channels to carry a number of signals given by the (N-1)th power of 2. So that 3 channels will carry 4 signals, 4 channels will carry 8 signals, 5 channels 16 signals, 6 channels 32 signals, 7 channels 64 signals, and so on.

You can see from this that where a small number of signals is involved, the advantages of multiplexing are quite small and may not be worth the effort. However, as the number of signals grows, the saving in channels rapidly becomes significant.

Although multiplexer and demultiplexer circuits may be built up from discrete logic elements using configurations like those shown in Figs. 2 and 3, many of the often-required circuits are in fact made available as single integrated circuits. Thus one can buy integrated 16-line to 1-line multiplexers in both TTL and CMOS varieties, for example, along with other types of multiplexer and demultiplexer.

Actually because of the similarity between decoders and demultiplexers some of the integrated devices are designed to be used as either. Known as decoder/demultiplexers, those devices are basically a full demultiplexer like that of Fig.

3. To use them as a normal decoder, the data input is merely taken permanently to the logic 1 level.

Although the multiplexers and demultiplexers we have looked at so far have been single-pole types, these are by no means the only type used. Quite a number of different varieties are used in practice; Fig. 4 shows another common type, a quad 2-line to 1-line multiplexer.

As you can see, this is basically the digital logic equivalent of a conventional 4-pole 2-position switch. The input lines A0-A3 are connected to the outputs Y0-Y3 in one "position", while the input lines B0-B3 are connected to the outputs in the other.

The multiplexer is controlled by a logic signal applied to the select input. When the select input is taken to the logic 1 level the A input lines are connected, and conversely when the select input is taken to the 0 logic level the B input lines are connected.

This particular type of multiplexer is not particularly useful for data communication applications, as it would only be capable of multiplexing two groups of four signals over a single set of four channels. However multiplexers of this type are very useful in applications where a number of sources of multi-bit data must be fed selectively into a digital register, logic circuit or arithmetic unit. In such applications the multi-pole multiplexer is often described as a "data selector."

A simple illustration of this type of application is shown in Fig. 5. Here an 8-bit register with parallel loading inputs is fed from an octal 2-line to 1-line multiplexer, used to select one of two alternative sources of 8-bit data: A and B.

If the parallel load control line of the register is pulsed or "strobed" while the select input of the multiplexer is held at the logic 1 level, the register will load the data from the A source. If on the other hand the multiplexer select input is held at the logic 0 level when the register parallel load line is strobed, the data will be loaded instead from the B source.

This sort of circuit is used quite frequently in digital computers, where data must often be loaded into various registers from a variety of other registers and logic circuits.

Another important application of multipole multiplexers is illustrated in Fig. 6. This shows a multiplexed display system, of the type used in many situations where more than three or four numerical digits or alphanumeric characters must be displayed at nominally the same time. Typical applications are in electronic calculators, digital watches and clocks, and digital measuring instruments like frequency counters.

As you can see, Fig 6 actually shows an 8-digit display using 7-segment readouts. If multiplexing were not used such a display would require eight separate BCD-to-7-segment "decoders", with segment drivers, one for each display. But as you can see, by multiplexing the eight displays we reduce this requirement to a single decoder-driver.

The multiplexing is done in this case by a quad 8-line to 1-line multiplexer, which is the equivalent of a 4-pole 8-position switch. This has eight sets of four inputs, with each set connected to a source of BCD data from the calculator's arithmetic register or the frequency meter's counting decades. The outputs of the multiplexer Y0-Y3 are then fed to the single BCD-to-7-segment decoder/driver, which in turn feeds all of the corresponding segments of the displays.

The 3-bit line address for the multiplexer is generated by a 3-bit counter, driven by a clock oscillator. The line address is also fed to a 3-bit decoder, whose outputs are fed to a set of eight switching transistors. The transistors control the individual digit display devices, and are thus described as the "digit selectors".

In effect, the 3-bit decoder and the digit selector transistors act as the demultiplexer in the system. As the clock oscillator cycles the 3-bit counter to force the multiplexer to feed the eight sets of BCD data in turn to the 7-segment decoder and the display segment lines, the decoder and digit selector transistors make sure that only the correct display is enabled when the appropriate data is present on the segment lines.

Of course in a multiplexed display of this type, only one of the digits is actually being displayed at any one instant. If the clock oscillator used in the multiplexing system were to be run at a very low frequency, this would become quite obvious: the various digits would be seen to be displayed one after the other, in cyclic fashion.

In practice the clock oscillator is run at

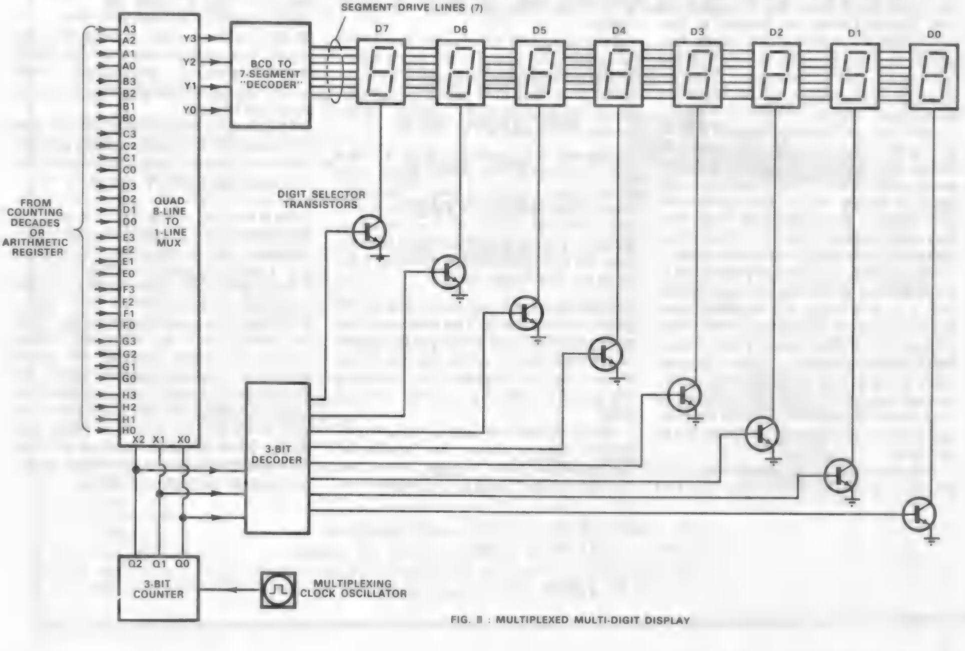

SEGMENT DRIVE LINES (7)
D7
D6
D5
D4
D3
D2
D1
D0
A3
A2
A1
A0
B3
B2
B1
B0
C3
C2
C1
C0
D3
D2
D1
D0
E3
E2
E1
E0
F3
F2
F1
F0
G3
G2
G1
G0
H3
H2
H1
H0
Y3
Y2
Y1
Y0
BCD TO 7-SEGMENT "DECODER"
QUAD 8-LINE TO 1-LINE MUX
FROM COUNTING DECADES OR ARITHMETIC REGISTER
DIGIT SELECTOR TRANSISTORS
X2
X1
X0
3-BIT DECODER
Q2
Q1
Q0
3-BIT COUNTER
MULTIPLEXING CLOCK OSCILLATOR

FIG. 6 : MULTIPLEXED MULTI-DIGIT DISPLAY

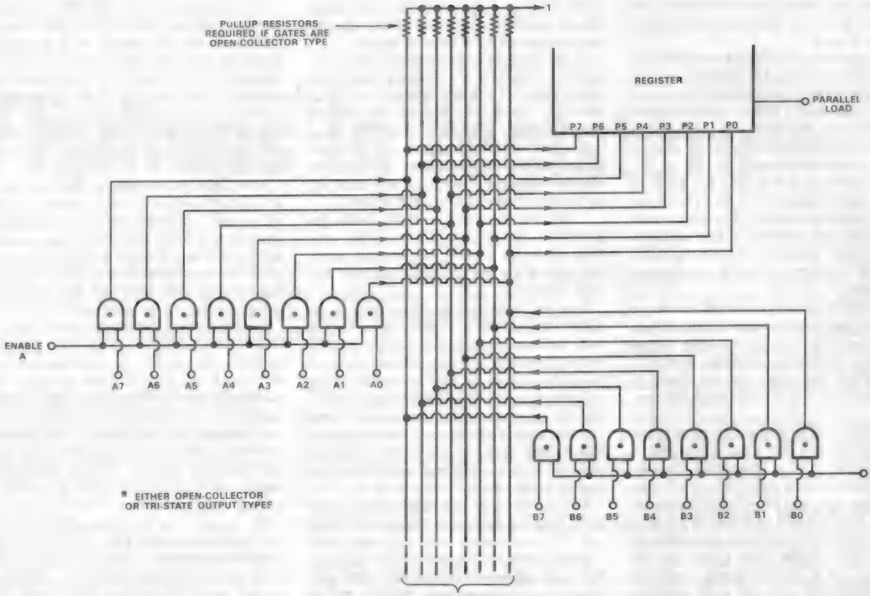


FIG. 7 : BUSLINE MULTIPLEXING

a frequency high enough to allow human persistence of vision to make it seem that all digits are being displayed continuously. Generally this means that each individual digit must be displayed no fewer than 50 times per second, so that the minimum multiplexing clock frequency is given by 50N, where N is the number of digits in the display. Thus for an eight digit display the multiplexing clock should be run at 400Hz or higher.

Apart from saving decoding and display driving circuitry, this type of multiplexed display can also give a saving in display power consumption. This is because some types of display device are significantly more efficient when operated under low-duty-cycle pulsed conditions, than when operated under steady-state conditions.

For example, a LED display operated in a multiplexed 8-digit display (and hence with a ⅛ duty cycle) at a segment drive current of 25mA is typically 75% more efficient than if the segments were operated at 5mA continuously. Hence the complete 8-digit display operating at an average segment current of 25mA will be 75% more efficient than if it were not multiplexed and operated at (8 times 5mA) or 40mA.

This means that to obtain the same brightness from a non-multiplexed display, it would probably have to be operated at about 8mA per segment per device, giving a total of (7 x 8 x 8mA) or 450mA total, compared with (7 x 25mA) or 175mA total for the multiplexed display.

Before we close this chapter, we should look briefly at a type of multiplexing which is now used quite extensively in computers and other data handling systems. This is the technique of "busline" multiplexing, illustrated in Fig. 7.

Basically this technique uses a "data bus" consisting of ■ set of signal lines which form virtually a data "highway" through the system. Data words are passed from one section of the system to another by feeding them onto the buslines from the source section, and picking them off at the destination section.

In essence, the buslines act as wired-OR gates (See chapter 6). The data words are multiplexed onto the lines by sets of gates, as shown in Fig. 7, while they are picked off and fed into registers by connecting the parallel load inputs onto the bus lines.

Hence to feed the data on lines A0-A7 into the register shown, the "enable A" line would be taken to logic 1 level and the parallel load input of the register pulsed. The "enable B" line would have to be kept at the logic 0 level during this operation, to ensure that the data on lines B0-B7 was not placed on the buslines as well.

Alternatively to load the data from B0-B7 into the register, instead, the "enable B" line would be taken to logic 1 level and the "enable A" line to logic 0 level, while the register's parallel load input was pulsed as before.

This type of multiplexing system may be expanded fairly easily to cope with virtually any number of input sources and output destinations. All that is necessary is to ensure that only one set of source gates is enabled at any one time, and that the parallel load inputs of the destination register(s) are enabled when the data they are to receive is present on the buslines.

Note that if the gates used to control the data sources are of the "open-collector" or uncommitted output type, the buslines must be fitted with pull-up resistors as shown. However the pull-up resistors are not required if the gates used are of the "Tri-state" type originated by National Semiconductor. These gates provide active pull-up as well as active pull-down, when enabled, quite apart from having a third state in which the output is "floating" at high impedance.

Chapter 14

Binary Arithmetic

Much of the information handled by digital circuits and systems is in the form of numerical information or numbers—usually binary numbers. Often arithmetic must be performed on the numbers, and to understand the operation of arithmetic circuits you must be familiar with the concepts of binary arithmetic.

As we have seen in earlier chapters, digital circuits and systems are often used to handle numerical information in the form of binary numbers. In circuits which form part of systems such as computers and calculators, it is often necessary to perform arithmetic operations on the information while in this form. In other words, to perform binary arithmetic.

Before we have a look at some of the circuits used to perform binary arithmetic, it may be worthwhile looking at the basic concepts of the arithmetic itself. While in many ways this is similar to the familiar decimal arithmetic, there are differences which can be confusing if not explained.

Let us first look at addition. This is actually rather simpler than decimal addition, as there are only four basic possibilities when two binary digits are added together:

$$\begin{array}{l} 0 + 0 = 0 \\ 0 + 1 = 1 \\ 1 + 0 = 1 \\ 1 + 1 = 0 \text{ with 1 to carry} \end{array}$$

These "four rules of binary addition" replace the usual decimal addition tables, but apart from this the addition of two binary numbers is carried out in a very similar manner to decimal addition.

Digits of the same binary weighting are added together, starting with the least significant digits on the right and working up to the most significant digits on the left—taking any carries into account.

This is best shown by an example. Here is the binary addition of 1101, equivalent to decimal 13, and 110 which is equivalent to decimal 6:

$$\begin{array}{r} 1101 + \\ 110 \\ \hline 10011 \end{array}$$

You can see that the addition of the least significant bits produces a sum of 1, with none to carry. This also occurs with the next significant bits. However the third pair of bits are both 1's so they produce a sum of 0 with a carry. The carried bit and the most significant bit of the upper number (the "augend") then add in similar fashion to produce a sum of 0, with a further 1 carried. This carry then becomes the most significant digit of the answer, 10011.

If you care to check the answer, you will find it is equivalent to decimal 19—the usual result when 13 and 6 are added together!

Here is a second example of binary addition, using two larger numbers:

$$\begin{array}{rl} 110111 + & \text{(augend)} \\ 1001 & \text{(addend)} \\ \hline 1000000 & \text{(sum)} \end{array}$$

The augend here is equivalent to decimal 55, and the addend to decimal 9. Notice that in this case a carry bit is generated as soon as the two least significant bits are added, and that this carry in effect generates further carries as each further pair of bits is added. Only when the addition reaches the end of the augend does the carry-over cease, to produce a sum of 1000000. This is equivalent to 64, the correct answer.

By the way, the augend is merely the "first" number which we take, while the addend is the number added to it. The augend is not necessarily the larger of the two, as this third example shows:

$$\begin{array}{rl} 1110 + & \text{(augend)} \\ 1100001 & \text{(addend)} \\ \hline 1101111 & \text{(sum)} \end{array}$$

This example also shows that it is quite possible to have an addition without any carries being generated at all. Here the two numbers are equivalent to decimals 14 and 97, giving a sum equivalent to decimal 111.

How about binary subtraction? Well, the rules for this are also fairly simple. Again there are only four:

$$\begin{array}{l} 0 - 0 = 0 \\ 1 - 0 = 1 \\ 0 - 1 = 1 \text{ but with 1 borrowed} \\ 1 - 1 = 0 \end{array}$$

Using these basic rules as a guide, binary subtraction may be preformed in much the same manner as decimal subtraction. Digits of the same binary weighting are subtracted from each other, starting as before with the least significant bits and working up to the most significant—this time taking borrows into account.

Here is a simple example of binary subtraction performed in this so-called "direct" way, where the first number is the "diminuend" and the "subtrahend" is the number subtracted from it:

$$\begin{array}{rl} 10110 - & \text{(diminuend)} \\ 101 & \text{(subtrahend)} \\ \hline 10001 & \text{(remainder)} \end{array}$$

Here when the two least significant bits are subtracted, there is a remainder of 1 but 1 has to be borrowed. This gives a borrow bit to be subtracted from the next bit of the diminuend, giving a remainder of 0 with no borrow. Then both of the third bits are 1, so there is again a remainder of 0 with no borrow. Finally there are no fourth and fifth bits in the subtrahend, so these bits of the diminuend are effectively transferred into the remainder.

The remainder is in this case equivalent to decimal 17, which is the correct answer as the diminuend is equivalent to decimal 22 and the subtrahend to decimal 5.

Although this example is quite straightforward, direct binary subtraction tends to become rather tricky when adjacent zeroes in the diminuend cause cumulative borrowing. This is shown in the following example:

$$\begin{array}{rl} 10001 - & \text{(diminuend)} \\ 111 & \text{(subtrahend)} \\ \hline 01010 & \text{(remainder)} \end{array}$$

Here the two least significant bits subtract to give a remainder of 0, with no borrowing. But the next bit of the diminuend is 0, so a borrow must be made to allow subtraction of the second subtrahend bit. The remainder of the second bit-pair subtraction is thus 1.

When we come to consider the third bit-pair, it starts getting tricky. We already have a borrow from the second pair subtraction, but the third diminuend bit is already 0. So we must again borrow, effectively giving binary "10" (equivalent to decimal 2) in the diminuend bit position. From this we must subtract both the borrow 1 and the subtrahend 1, leaving a remainder of 0.

There is no subtrahend bit in the fourth position, but we have a borrow from the third pair subtraction. However the diminuend bit is again 0, so we must borrow yet again. This effectively gives us "10" again in the diminuend, from which the previous borrow 1 must be subtracted to give a remainder of 1. Finally the borrow from this fourth subtraction must be subtracted from the most significant diminuend bit, to give a 0 in the remainder. Whew!

Incidentally the decimal equivalents of the diminuend and the subtrahend in this example are 17 and 7 respectively, while the remainder is equivalent to the correct answer of decimal 10.

But the real problem with the direct method of binary subtraction from a practical point of view is that it cannot cope easily with situations where the subtrahend may be larger in magnitude than the diminuend, giving a negative remainder.

To illustrate this, consider the following example where the diminuend is equivalent to decimal 7, and the subtrahend to decimal 9:

```
    0111 — (diminuend)
    1001   (subtrahend)
 (—1)110   (remainder)
```

The subtraction of the least significant bit pair is straightforward enough, with a remainder of 0 and no borrowing required. The next bit pair are also straightforward, this time with a remainder of 1 and no borrowing. And the third bit pair are the same. But what happens with the fourth or most significant bits?

Here we are in trouble, because in order to produce a normal positive remainder, we would need to borrow. But there is nothing left to borrow from!

In effect, then, we are left with a remainder of —1 in the most significant bit position, as shown. And in this case the bit position concerned has a binary weighting of 8, so that we have a two-part remainder: the positive part, corresponding to decimal 6, and the negative part corresponding to —8. To get the correct answer, equivalent to decimal —2, we would have to add these two parts of the remainder together algebraically.

Note that the need to do this extra step didn't actually become apparent until we got to the last pair of bits in the subtraction. Until then, it seemed completely normal and straightforward.

This is what causes problems when it comes to performing the direct method of binary subtraction with digital arithmetic circuits. If a subtractor circuit is to be able to cope with situations where the subtrahend may be larger in magnitude than the diminuend, it must not only be provided with the facility to perform algebraic addition of partial remainders, but also with the logic to detect when this is to be done. To do these tasks efficiently requires quite complex circuitry.

As it happens, there is a very much simpler and more convenient way around the problem. Instead of performing direct subtraction, the same result is achieved by performing an equivalent operation: complementary addition.

Complementary addition relies on the fact that subtracting a number B from another number A is basically the same as adding minus-B to it:

$$A - B = A + (-B)$$

In other words we can perform subtraction by straightforward binary addition, merely by converting the subtrahend number beforehand into its negation or complement.

The most common type of complementary addition uses the so-called "two's complement" of the subtrahend number.

The two's complement of a binary number is defined as the difference between the number and the base 2 raised to the same power as the total number of bits being used, i.e.,

$$C = 2^b - N$$

where N is the number concerned, C is its two's complement, and b is the number of bits being used to represent them.

This may seem a bit mysterious, but an example should make it more clear. In many digital systems, 8-bit words are used to represent numbers, so in such systems the two's complement of a number is defined as the difference between the number and two raised to the power of 8, or the equivalent of 256 decimal (100000000). Hence in an 8-bit system, the two's complement of a number N is given by:

$$C = 100000000 - N$$

where both C and N are in binary notation. Note that as the eighth power of two is actually a 9-bit number, it is effectively the same as zero in the eight-bit system concerned (the ninth bit would be ignored). So that this expression can in fact be rewritten as

$$C + N = 00000000$$

or in other words, the two's complement of a number is that number which when added to the number produces a sum of zero.

Even this is not all that convenient from a practical point of view, as you'll find if you try to work out a few specific examples. However as it happens there is a quite easy way of finding the two's complement of a binary number: complement each bit individually, and then add 1 to the resulting number.

TABLE 1

Decimal	Binary (2's Complement)
16	00010000
15	00001111
14	00001110
13	00001101
12	00001100
11	00001011
10	00001010
9	00001001
8	00001000
7	00000111
6	00000110
5	00000101
4	00000100
3	00000011
2	00000010
1	00000001
0	00000000
—1	11111111
—2	11111110
—3	11111101
—4	11111100
—5	11111011
—6	11111010
—7	11111001
—8	11111000
—9	11110111
—10	11110110
—11	11110101
—12	11110100
—13	11110011
—14	11110010
—15	11110001
—16	11110000

Here are a couple of examples to illustrate this. First, the 8-bit two's complement of the binary equivalent to decimal 5:

```
    Original number: 00000101
Complement each bit: 11111010
         Then add 1: 11111011
```

And the 8-bit two's complement to the equivalent to decimal 73:

```
    Original number: 01001001
Complement each bit: 10110110
         Then add 1: 10110111
```

As you can see it is fairly easy to find the two's complement of a number using this method.

To help you visualise two's complement notation a little more clearly, table 1 shows the 8-bit binary equivalents to the decimal numbers from 1 to 16, together with those for their two's complements. The equivalent to 0 is also shown, for reference.

If you care to work it out, you will find that the two's complement notation for —1 is basically the same as that for 255 in normal or unsigned binary notation. Similarly that for —2 is the same as 254, that for —3 the same as for 253, and so on. So that the negative numbers are effectively "working down" from 256, as they should where 8-bit two's complement numbers are concerned.

Fairly obviously, the negative numbers can't "work down" indefinitely, because they would become indistinguishable from the positive numbers. In fact to allow the two to be distinguished unambiguously, an arbitrary convention is adopted, to divide the total number of available bit combinations into two roughly equal groups—one for the positive numbers, and the other for negative numbers.

The convention adopted is that numbers having a 0 in the most significant bit position are interpreted as being positive, while those with a 1 in the most significant bit position are interpreted as negative.

So that with 8-bit numbers, the 256 available bit combinations are divided into 127 positive numbers, zero, and 128 negative numbers. The positive numbers range from 1 to the equivalent of decimal 127, or in binary from 00000001 to 01111111; while the negative numbers range from —1 to the equivalent of decimal —128, or in binary from 11111111 to 10000000.

Using two's complement notation, subtraction is performed in exactly the same manner as addition. The only difference is that the subtrahend is converted into its two's complement before the addition is carried out.

If we wish to subtract the equivalent of decimal 7 from the equivalent of decimal 25, for example, this is done in the following way. First, the binary equivalent of 7 is converted into its two's complement:

Original: 00000111
Bits Complemented: 11111000
1 added: 11111001

The two's complement is then simply added to the equivalent of 25, with any carry-over from the eighth bit addition ignored since we are working with 8-bit numbers:

Augend (25): 00011001
Addend (—7): 11111001
Sum (18): 00010010

With two's complement arithmetic, there is no problem in coping with subtractions where the subtrahend may be larger than the diminuend, giving a negative result. All that happens is that the answer has a 1 in its most significant bit position, signifying that it is negative.

This is shown by the following example, where the equivalent of decimal 7 is added to the two's complement equivalent of —43. In effect, this is the same as subtracting 43 from 7:

Augend (7): 00000111
Addend (—43): 11010101
Sum (—36): 11011100

From a practical point of view, the advantage of using two's complement notation is that both addition and subtraction may be performed by a relatively straightforward binary adder circuit. Most binary arithmetic circuits are thus designed to handle numbers in two's complement notation, and are described as performing "two's complement arithmetic".

So far, we have considered only addition and subtraction of binary numbers. In fact these are the arithmetic operations most often performed, as multiplication and division are often performed by repetitive addition or subtraction methods.

Binary multiplication is at times performed by the "traditional" approach, particularly when numbers must be multiplied as rapidly as possible. So we should perhaps look briefly at the basic concept.

As with addition and subtraction, there are again only four basic "laws" where multiplication is concerned:

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Using these as a guide, the actual multiplication of two binary numbers may be performed in a similar manner to decimal multiplication. The multiplicand is multiplied separately by each bit of the multiplier, giving a series of partial products which are then added together to give the final product.

This should be clear from the following example, which shows the equivalent of decimal 22 multiplied by the equivalent of decimal 5:

```
  00010110×  (multiplicand)
  00000101   (multiplier)
  --------
  00010110
 00000000    (partial products)
00010110
----------
  01101110   (final product)
```

Note that only the first three partial products are shown, as the remainder are all zero in this example. Note also that only the eight least significant bits of the final product are shown, as these would the only bits used in an 8-bit system.

More importantly, though, note that the partial products are obtained much more simply than in the case of decimal multiplication. Where the multiplier bit concerned is a 0, the partial product is also zero; where it is a 1, the partial product is simply the multiplicand displaced to the left by the appropriate number of bit positions.

This illustrates an important point about binary numbers: shifting a number one place to the left is equivalent to multiplying by 2, while shifting one place to the right is equivalent to dividing by 2. This corresponds exactly to decimal notation, where a shift to the left is equivalent to multiplying by 10, and a shift to the right to dividing by 10.

High-speed binary multiplier circuits tend to use this as a way of performing multiplication by the traditional method we have just illustrated. The partial products are obtained by repetitive shifting of the multiplicand to the left, and each time using the appropriate bit of the multiplier number to determine whether the partial product is added to the accumulating final product.

As mentioned, binary multiplication is very often done not by this traditional method, but by repetitive addition. This simply relies on the fact that multiplication of one number by another is equivalent to adding the multiplicand to itself a number of times, equal in number to the multiplier.

Hence an alternative approach to multiplication is to arrange that the multiplicand number is added to itself in a normal binary addition circuit, and each time an addition is performed the multiplier number is decreased by 1 (decremented). The process is arranged to stop when the multiplier reaches zero, whereupon the multiplicand must have been added to itself the correct number of times.

This is the method of multiplication used by most minicomputers, microprocessors and digital calculators, as we shall see in a later chapter.

Binary division is generally performed in a similar fashion, except that repetitive subtraction is used instead of addition. As before the subtraction is generally done using two's complement addition.

Before we leave the topic of binary arithmetic for the time being, there are a few points which should perhaps be made.

As we have seen, it is very convenient to use two's complement notation in performing binary arithmetic. This involves using one bit of the word length available—the most significant bit—to indicate the sign of each number.

One result of this is that a given word length can only be used to represent half as many absolute magnitudes as with unsigned binary notation. For example with unsigned notation, 8-bit words can represent the numbers from 0 to the equivalent of decimal 255, whereas with two's complement signed notation they can only represent absolute magnitudes from 0 to 127 in the positive direction, and from 0 to 128 in the negative direction.

Of course there is the added advantage of being able to represent negative numbers as well as positive, so that the information-carrying efficiency is not impaired. But despite this, there are applications where the halving in the absolute magnitude range is significant.

One way of getting around this is to use longer words to represent the numbers. Hence if 16-bit words are used instead of 8-bit words, with two's complement notation the range of numbers which can be represented increases significantly: from the equivalent of —32,768 decimal, to the equivalent of 32,767. With 24-bit words the range expands much further again, equivalent to the decimal range from —8,388, 608 to 8,388,607.

Some large digital systems use 16-bit, 24-bit or even 36-bit words throughout, in order to have the necessary magnitude range and arithmetic resolution. However this tends to be quite costly, as the complete system must be capable of handling the longer words.

With smaller systems, an alternative approach is usually adopted. Here the system is designed to handle relatively small words—say 8 bits in length—but when required the arithmetic circuitry can effectively handle larger numbers by dealing with them in "multiple word chunks".

This is known as the technique of multiple precision. Most commonly, either double precision or triple precision are used, which as the names suggest involve using either two or three separate words to represent each number.

Hence if an 8-bit system is said to be capable of performing double-precision arithmetic, this simply means that its arithmetic circuits are effectively able to perform arithmetic on 16-bit numbers, handling them in two separate steps: first the 8 least significant bits, and then the 8 most significant bits.

Similarly an 8-bit system capable of performing triple-precision arithmetic has arithmetic circuits effectively capable of performing arithmetic on 24-bit numbers, handling them in three separate steps.

As you might imagine, the advantage of multiple precision techniques is that most of the digital system concerned is only required to handle relatively small words. Even the multiple-precision arithmetic circuits themselves may be somewhat simpler than circuits designed to handle the larger words, if the arithmetic is actually performed in separate steps. All that may be required is a flipflop or two to store carry-over from one group of bits to the next, and perhaps a partial product storage register in the case of a multiplier.

Chapter 15

Arithmetic circuits

Following on from the discussion of binary arithmetic given in the previous chapter, we are now in a position to look at the actual circuit configurations used to perform arithmetic in practice. The main type of circuit discussed is the binary adder, widely used to perform not just addition, but most of the other basic arithmetic as well.

As we saw in the last chapter, the adoption of two's complement notation allows binary subtraction to be performed in exactly the same way as addition. It is also possible to perform binary multiplication by repetitive addition, and division by repetitive subtraction. Hence a circuit designed to perform binary addition may be used to perform virtually all of the basic arithmetic functions, and this is very often done.

From a practical point of view, this means that binary "adder" circuits tend to form the backbone of digital arithmetic hardware. So that if you have a basic understanding of adders and the way they are used, you should find it fairly easy to find your way through most arithmetic systems.

This being the case, we will confine ourselves almost completely in this chapter to a discussion of binary adders and the "arithmetic-logic units" or ALUs which have been developed from them.

You may recall from the previous chapter that there are only four basic possibilities when two digits or "bits" are added together. We described these as the "four rules of binary addition".

To produce a binary adder circuit, we must basically produce a circuit which responds to two input bits according to these rules. In other words, we must produce a circuit whose behaviour corresponds to the truth table shown in Table 1. If you compare this table with the "four rules" given in the previous chapter, you'll see that it simply presents the same information in slightly different form.

TABLE 1

INPUTS		OUTPUTS	
ADDEND A	AUGEND B	SUM Σ	CARRY C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

As you can see there are two outputs, one the SUM output (usually symbolised as shown by the Greek sigma) and the other the CARRY output.

In short, then, a basic binary adder is a circuit with two inputs and two outputs, which functions so that the outputs have the truth-values shown for the various truth-value combinations of the two inputs.

If you look at Table 1 fairly closely, it should become clear that in fact the SUM output corresponds to a logic exclusive-OR or EX-OR function between the two inputs, while the CARRY output corresponds to a simple AND function.

There are quite a few different logic configurations which will perform these functions. A fairly simple example is shown in Fig. 1, using 3 AND gates, two inverters and an OR gate.

You have probably noticed already that this is actually labelled a "half-adder" circuit. This is because in considering binary addition so far, we have in fact only been looking at a special case: where we are adding the least significant bits (LSBs) of two numbers. It is really only when LSBs are added together that the addition involves only two bits.

More generally, there are in fact three bits to be added together: the addend bit, the augend bit, and a carry-over bit from the addition of the next less significant bit pair.

In other words a full binary adder circuit must be able to cope with three inputs, and have ■ truth table corresponding to that shown below in Table 2.

Again, there are quite a few different logic configurations which will perform these functions. An example is shown in Fig. 2. As you can see it is rather more complex than the half-adder of Fig. 1, using seven 3-input AND gates, three inverters and two 4-input OR gates.

There are two broad ways in which binary adders are used to perform addition of complete binary numbers, one using the serial or "bit-by-bit" approach and the other the parallel or "all bits at once" approach. The serial approach is relatively low in cost, but rather slow; the parallel approach is fast but tends to be much more costly.

TABLE 2

INPUTS			OUTPUTS	
CARRY C	ADDEND A	AUGEND B	SUM Σ	CARRY C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Typically, serial binary addition involves a single full-binary adder, through which the addend and augend numbers are passed bit-by-bit and corresponding bits together—starting with the LSBs. Usually the two numbers are shifted into the adder from ■ pair of shift registers, with the sum output of the adder fed to either a third shift register or back into one of the original registers (usually the addend register). A single flip-flop is used to store the carry-over.

The basic configuration for an adder of this type is shown in Fig. 3. Here the adder is designed to handle 8-bit numbers, and the addend and augend registers are 8 bits wide (i.e., they each consist of 8 flipflops). The addend register is also used to receive the sum, this being done by connecting the sum output of the adder back to the D input of the most significant bit (MSB) flipflop.

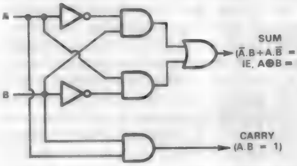


FIG. 1: HALF ADDER CIRCUIT

The carry flipflop is basically a D-type device, with its D input fed from the carry output of the adder, and its Q output connected to the carry input of the adder. The shift clock lines of both the addend and augend registers are connected together, and to the CLK input of the carry FF, to form a master clock pulse line.

In operation, the two numbers to be added together are first loaded into the addend and augend registers. The loading may be done either by parallel loading or serial shifting, as convenient, although the circuitry to do either has been omitted from Fig. 3 in the interests of clarity.

The carry flipflop is then cleared, to make sure that it does not inject a spurious carry-over from some previous addition, or from its random content after power-up.

At this stage, the A and B inputs of the adder will already have presented to them the LSBs of the two numbers, from the two register outputs. Hence the sum output of the adder will already represent the LSB sum of the two numbers, and the carry output any carry-over which should be generated.

Hence if a single clock pulse is applied to the clock line, three things will happen:

1. The carry FF will store the carry-over of this first bit-pair addition, ready for the next bit-pair addition.
2. The LSBs of both addend and augend will be shifted out of their respective registers, and lost, leaving the next significant bits applied to the adder inputs.
3. The LSB of the sum, produced by the adder, will be shifted into the MSB bit position of the addend register.

A second clock pulse causes this sequence of events to be repeated, except that in this case the sum of the second pair of bits is shifted into the MSB bit position of the addend register, and the LSB sum bit is shifted one place to the right. The second bits of the original addend and augend are lost as before, while the carry FF stores the new carry bit.

The complete addition of the two 8-bit numbers requires a total of 8 clock pulses. At the end of the eighth clock pulse, the addend register will contain the 8-bit sum of the two. The augend register will be cleared, as the augend will have been shifted fully out. The carry FF will also contain the end-carry, or carry generated by the final addition of the two MSBs. This may or may not be used, depending upon the rest of the system.

The eight clock pulses used to perform this addition process must be spaced at sufficient intervals to allow for the switching times of the register flipflops, the carry flipflop and the adder logic elements. Hence serial addition tends to take an appreciable time—and a time directly proportional to the bit length of the numbers involved, for a given type of logic. So that adding two 16-bit numbers will take twice as long as adding two 8-bit numbers, and so on.

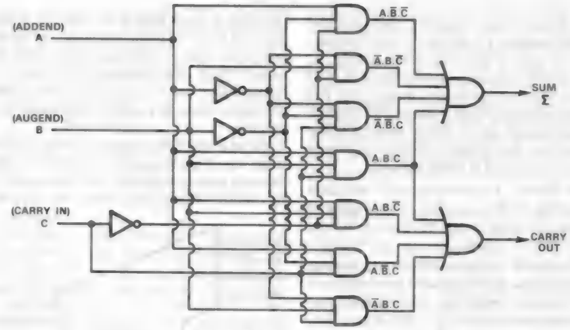


FIG. 2: FULL ADDER

Considerably faster addition can be performed using the parallel approach, because all bit pairs are added virtually simultaneously. This also makes the addition time largely independent of number size, although adders for large numbers can require special carry logic if high speed operation is to be achieved.

Basically, parallel addition involves a set of full adders working in parallel, one for each bit pair. This is illustrated in the diagram of Fig. 4.

Shown in this diagram is a 4-bit parallel adder, using four full adder circuits. The lowest adder takes the two LSBs of the two input numbers, AO and BO, and produces the appropriate sum sigma-0 and carry CO. The latter is then used by the second adder, together with bits A1 and B1 of the two input numbers, to produce the second sum sigma-1 and carry C1. The third and fourth adders work in the same fashion.

If a 4-bit parallel adder like that shown in Fig. 4 is used in a system dealing solely with 4-bit numbers, the carry input of the LSB adder would be disabled by connecting it to the logic 0 level as shown. Similarly the carry output of the MSB adder (C3) would probably be left unused. However the existence of this input and output does allow a number of such circuits to be used together, to handle larger numbers.

To form an 8-bit adder, for example, two such circuits would be connected together with the C3 output of the circuit handling the four least significant bits fed to the LSB carry input of the circuit handling the four most significant bits. To handle 16-bit number four circuits would be cascaded in the same way.

The circuit of Fig. 4 could thus be described as effectively providing a 4-bit ''slice'' of parallel adder circuitry, as it is really a building-block suitable for building a parallel adder of almost any desired size.

At this stage it might be worthwhile to point out that the addition of two numbers is by no means the only task that can be performed by adder circuits like those shown in Figs. 3 and 4. As you might expect, they can be used for two's complement subtraction, by forming the subtrahend number into its two's complement before an otherwise normal binary addition.

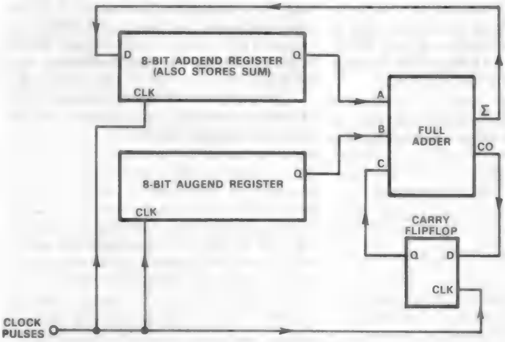


FIG. 3: 8-BIT SERIAL ADDER

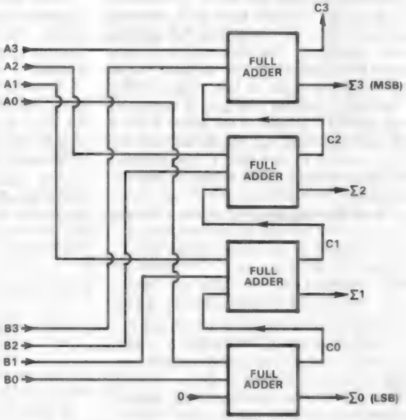


FIG. 4: 4-BIT PARALLEL ADDER

Apart from this, they can also be used for incrementing a single binary number. To do this with the serial adder of Fig. 3, for example, it is merely necessary to load the number to be incremented into the addend register, clear the augend register (or load it with zero), and preset the carry FF—rather than clear it—before applying the usual train of 8 clock pulses.

The initial "1" in the carry FF will thus be added to the number in the addend register, and the result returned as before to the same register.

This ability to serve as an incrementer may also be used to convert numbers into their two's complement, prior to performing subtraction.

As you may recall from the previous chapter, the most convenient way of generating the two's complement of a number is to complement each of the bits individually, and then add 1 to the resulting number. This is the method used here.

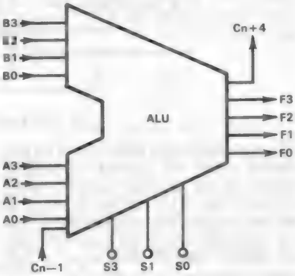
With a serial adder like that in Fig. 3, this would be most easily done by providing additional logic at the output of the addend register, so that by the application of a control signal the A input of the adder could be derived from the Q-bar output of the LSB addend flip-flop, instead of the Q output. This would then mean that when a number is shifted out of the addend register into the adder, it comes out in bit-complemented form—also known as the "one's complement".

So that by loading the number to be complemented into the addend register, clearing the augend register, and presetting the carry FF before we apply the usual 8 clock pulses, we can form the two's complement of the number in a single operation. The logic at the output of the addend register causes the number to be fed to the adder in one's complement form, while the initial "1" in the carry FF causes it to be incremented into the two's complement.

Having formed the two's complement in this way, it would then be a relatively simple matter to complete a full subtraction operation. The diminuend number would be loaded into the "augend" register, and the logic at the output of the "addend" register changed to reconnect the normal Q output to the adder A input. Performing a normal binary addition will then result in the two's complement remainder being left in the "addend" register.

Note the sequence of events: the subtrahend is dealt with first, loading it into the "addend" register and then using the adder to turn it into the two's complement. Then the diminuend number is loaded into the "augend" register, and the adder used to perform the actual subtraction.

As you can see, a binary adder can be used for other things apart from straightforward binary addition. In fact by adding a bit of additional circuitry here and there, it becomes possible to perform quite a number of different logical and arithmetic operations.



SELECT CODE			FUNCTION PERFORMED BY ALU
S3	S1	S0	
0	0	0	Fn = 0
0	0	1	Fn = An OR Bn
0	1	0	Fn = An.Bn
0	1	1	Fn = An ⊕ Bn
1	0	0	Fn = $(\overline{An})+1$
1	0	1	Fn = $(\overline{Bn})+1$
1	1	0	Fn = (An)+1
1	1	1	Fn = An+Bn+Cn—1

FIG. 5: 4-BIT PROGRAMMABLE ARITHMETIC-LOGIC UNIT (ALU)

When this is done the circuit tends to turn from an adder into a general-purpose programmable arithmetic-logic unit, or "ALU".

Such ALU circuits are widely used in modern digital systems; they are made as single LSI integrated circuits, and thus become important system building blocks.

Virtually every computer has such an ALU element forming the central part of its CPU, or central processing unit. Similarly an ALU forms an integral part of every microprocessor and calculator IC.

The logic symbol used to represent a 4-bit parallel ALU is shown in Fig. 5, together with a table illustrating the variety of logic and arithmetic functions which may be available in the repertoire of a typical device. The functions performed are selected by control logic signals applied to three "function select" inputs, S0, S1 and S2. As the device thus accepts three bits of control information, it follows there are eight different functions available ($2^3=8$ selection possibilities).

As you can see, one of the eight possibilities is shown here as being a "device inactive" or "disabled" mode: when the select code is 000, the outputs of the device are all held at the zero logic level.

All of the remaining seven possibilities are used to provide useful logic and arithmetic operations. There are three logic functions, with the code 001 producing the OR function, the code 010 producing the AND function and the code 011 producing the exclusive-OR function.

The remaining four codes produce arithmetic functions. Code 100 results in two's complementing of the input number A, and code 101 produces the same effect on number B. Code 110 results in number A being incremented, while the final code 111 produces full binary addition of the two numbers complete with any lower-order carry via the input Cn-1.

As with the parallel adder of Fig. 4, the ALU of Fig. 5 is designed to be suitable for use in multiples to handle larger numbers. It could thus be described as a "4-bit slice" of an ALU, or as a "4-bit ALU slice". Two such slices could be used to handle 8-bit numbers, or four to handle 16-bit numbers, etc.

In this discussion of binary arithmetic circuits we have looked mainly at adder circuits, and we have also considered only pure-binary arithmetic. Hopefully this simplification has made it easier for you to grasp the basic concepts involved, so that you should now be in a position to understand other types of arithmetic circuits when you come across them.

Pure binary arithmetic based almost entirely on adders or ALU elements is now used in a majority of digital computers—certainly in all microprocessors, and in most minicomputers. Some of the larger computers perform arithmetic in a BCD code instead of pure binary, and this is also done in most digital calculators. Larger computers also tend to use additional types of arithmetic circuitry to supplement adders: circuits like fast multipliers and dividers.

Unfortunately these circuits and those for BCD arithmetic are too specialised for this introductory course.

Chapter 16

Timing & Control

The operation of most digital systems is dependent upon a train of clock pulses fed to the various circuits to co-ordinate their functions. Other timing and gating signals may also be required, to initiate and control operation. In this chapter we look at some of the circuits used to generate these timing and control signals.

The operations performed by the various circuit sections within a digital system must often be either synchronised with one another, or arranged to take place according to a specific time sequence, to ensure the correct operation of the system as a whole. Usually the most efficient and convenient way of achieving this synchronism and/or time sequencing is to generate a train of pulses which are fed throughout the system to provide a timing reference. Logically enough these pulses are called "clock" pulses, and the circuits which generate them are known as clock generators or clock oscillators.

Quite often other timing signals are derived from the clock pulses, to further control system operation. In addition the clock pulses themselves are often used within the various circuits of the system, to ensure correct internal operation. Thus as we saw in chapters 9 and 13, the operation of both shift registers and multiplexers depends quite fundamentally upon a train of clock pulses.

In short, most digital systems include at least one clock generator, whose output pulses play a key role in overall system operation.

In many systems, the clock pulses do not have to be especially stable in terms of width or repetition rate. For such systems almost any oscillator capable of producing reasonably fast rectangular pulses may be used as a clock generator. However, there are some systems where timing stability is somewhat more critical; for these it is generally necessary to use a quartz crystal oscillator as the clock generator.

Whether the clock generator is crystal controlled or not, it is generally designed to use logic elements of the same "family" as used in the rest of the system concerned. So that if the system is based primarily on TTL devices, the clock generator will normally use TTL elements also. Similarly a clock generator for a system using CMOS and/or other MOS devices would normally use CMOS elements.

This is partly a matter of convenience, although not entirely so. If it uses elements from the same logic family as the rest of the system, the clock generator will be able to operate from the same supply voltage rails. Normally this will also ensure that its output pulses are compatible with the requirements of the rest of the system in terms of voltage swing, rise and fall times, and so on.

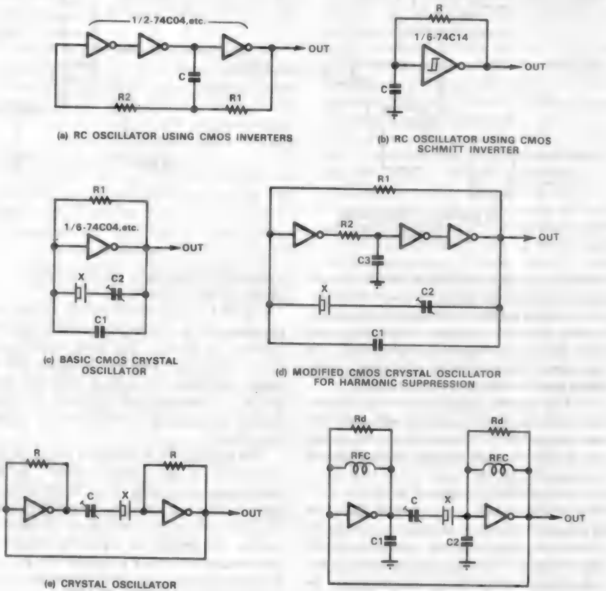
Of course not all systems use a single logic family throughout. Some use two or more families, and in these cases it may not be possible to design the clock generator so that its output pulses are directly compatible with all parts of the system. Special "level translator" elements may have to be employed, not just for the clock pulses, but also for any other signals which must pass between devices belonging to different families.

This is not always the case—many NMOS and PMOS devices are provided with internal level translators, so that they may be interfaced with TTL devices directly. Thus a system using some of these devices in conjunction with TTL devices can generally use a TTL clock generator, with no complications.

Some logic families are also partly compatible. For example the logic levels provided by CMOS elements are generally compatible with TTL inputs, although the reverse is not true. So that providing loading considerations are taken into account, a system using both CMOS and say low-power Schottky TTL devices can often be driven directly by a clock generator which uses CMOS devices, without level translators.

Let us now leave these general considerations and look at some specific clock generator circuits. Some representative circuits are shown in Fig. 1.

A simple RC clock oscillator using CMOS inverter elements is shown in (a). This type of oscillator relies for its operation on the propagation delay through



(a) RC OSCILLATOR USING CMOS INVERTERS

(b) RC OSCILLATOR USING CMOS SCHMITT INVERTER

(c) BASIC CMOS CRYSTAL OSCILLATOR

(d) MODIFIED CMOS CRYSTAL OSCILLATOR FOR HARMONIC SUPPRESSION

(e) CRYSTAL OSCILLATOR USING TTL INVERTERS

(f) MODIFIED TTL CRYSTAL OSCILLATOR FOR IMPROVED MODE STABILITY

FIG. 1 : CLOCK OSCILLATORS

each inverter—the finite time taken before a change in the input logic level is reflected in a change in the output logic level. Because of this delay, a loop containing an odd number of inverters has a natural tendency to oscillate; in effect, a logic "1" chases itself around the loop.

While a simple loop of inverters will generally oscillate, the frequency of oscillation tends to be rather unpredictable. It also tends to be heavily dependent upon supply voltage and temperature, making such an oscillator rather unsuitable as a clock generator. However if an RC circuit is added as shown, to introduce a delay which is relatively large compared with that of the inverters themselves, the frequency of oscillation becomes both more predictable and more stable.

In fact for the CMOS inverters shown, and if R1 is made equal in value to R2, the frequency of oscillation of the circuit in Fig. 1(a) is quite closely defined by the expression

F = 0.559/RC where R = R1 = R2. The duty cycle of the oscillations produced by this circuit is close to 50%—i.e., it produces a nominal squarewave output. The stability against power supply and temperature variations can be quite good, especially at low frequencies where R and C become relatively large.

Fig. 1(b) shows an even simpler oscillator, again of the RC type but in this case using a single CMOS Schmitt trigger inverter element. Here operation depends upon the fact that a Schmitt trigger element exhibits "hysteresis": its input switching threshold when the input level is rising is different from that when it is falling, unlike the thresholds of a normal inverter or gate element which are the same in both directions.

Schmitt trigger elements are named after Otto H. Schmitt, who first described a circuit displaying this type of behaviour in 1938. With a Schmitt trigger, the input threshold VTH where the trigger switches when the input voltage is rising from the low to high logic levels is significantly higher than the threshold level VTL where the trigger switches back again when the input voltage falls from the high to low logic levels. It is the difference between VTH and VTL which is defined as the hysteresis.

For a 74C14 element as shown, the hysteresis is typically equal to about 0.4 of the supply voltage.

If an R-C integrator circuit is connected between the input and output of a Schmitt inverter as shown, the time delay combines with the hysteresis to make the inverter unstable. Its output switches back and forth between the high and low logic levels, producing a squarewave output waveform.

The stability of this simple clock oscillator tends to be quite modest, as the switching thresholds of a CMOS Schmitt element do not remain constant proportions of the supply voltage as the latter is varied. However, for systems which do not require the clock frequency to be particularly stable, one could scarcely find a simpler clock generator.

Even when a system requires the higher frequency stability of a quartz crystal clock generator, this does not necessarily mean that a complex circuit is involved. Fig. 1 (c) shows a basic CMOS crystal oscillator, based on a single inverter element, and as you can see it is very straightforward.

Resistor R1 is used to self-bias the inverter so that it operates on the linear part of its transfer characteristic, as an amplifier rather than a switch. To produce oscillations the crystal X is then connected between output and input, with capacitors C1 and C2 provided to allow vernier frequency adjustment. Shunt capacitor C1 is used to lower the frequency slightly, while series capacitor C2 is used to increase it slightly.

Resistor R1 is typically quite large, between 1 and 10 megohms.

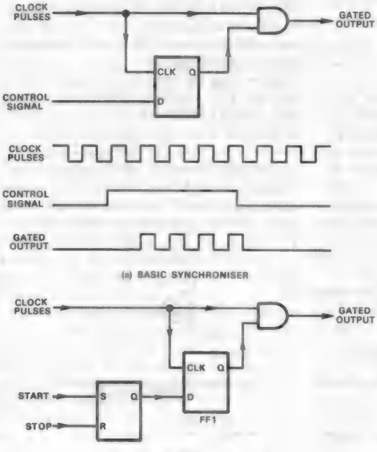
Although very simple, this oscillator gives very stable output at frequencies where the propagation delay of the inverter is small compared with the period of oscillation. Even for supply voltages as low as 3V this generally allows the circuit of Fig. 1(c) to operate reliably up to about 9MHz.

As with other simple crystal oscillators, there is a tendency for low frequency crystals to oscillate in an overtone mode rather than the desired fundamental. To prevent this, the propagation delay of the oscillation loop can be deliberately increased as shown in Fig. 1 (d). Here R2 and C3 have been added to increase the loop delay, while two more inverters have been added to maintain loop gain.

A simple crystal clock oscillator for TTL systems is shown in Fig. 1 (e). As you can see it uses two inverter elements, with self-biasing resistors connected across them to produce linear amplification. The two are connected in a loop, completed by crystal X and its series trimmer capacitor C.

The inverters may be either part of a 7404 or similar device, or inverter-connected gates from any of a variety of TTL devices. The biasing resistors R typically have a value of around 470 ohms.

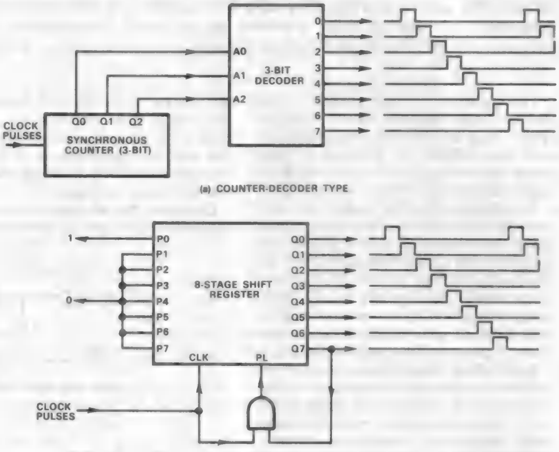
Like the circuit of Fig.1 (c), this simple circuit can allow crystals to oscillate in unwanted modes. It can also display reluctance to oscillate, with crystals of low activity. For these reasons designers sometimes use a modified configuration,



(a) BASIC SYNCHRONISER

(b) START-STOP SYNCHRONISER

FIG. 2 : PULSE SYNCHRONISERS



(a) COUNTER-DECODER TYPE

(b) SHIFT REGISTER TYPE

FIG. 3 : PULSE DISTRIBUTORS

such as that shown in Fig. 1(f).

Here capacitors C1 and C2 are used to lower the loop impedance as seen by the crystal, ensuring that the crystal oscillates in its primary mode. The biasing resistors are also replaced by RF chokes, designed to maintain loop gain at high frequencies so that reliable oscillation will occur with low activity crystals. Damping resistors Rd are used to prevent the circuit from oscillating at the self-resonant frequency of the RF chokes.

Having looked at how clock pulses may be generated, let us now look at some of the ways the clock pulses may be directed through a system and used to derive other control and timing signals.

Very often, a specific number of clock pulses are required for the correct operation of a circuit. For example, if an 8-bit binary number is to be shifted into or out of an 8-stage shift register, precisely 8 clock pulses will be involved—no more, and no less. And for correct operation each pulse must generally be complete and undistorted.

As you might perhaps expect, the basic way of deriving a specific number of clock pulses from the main pulse train is by gating. However, in order to make sure that the pulses gated out are not distorted, the gate must be opened and closed at suitable times—i.e., "between pulses". The gating must thus be **synchronised**.

Where the control signal used to operate the gate is itself derived from the clock pulses, this tends to occur automatically. However, there are many occasions where pulses must be gated in response to a non synchronised or "random" input, such as a signal arriving from outside the system.

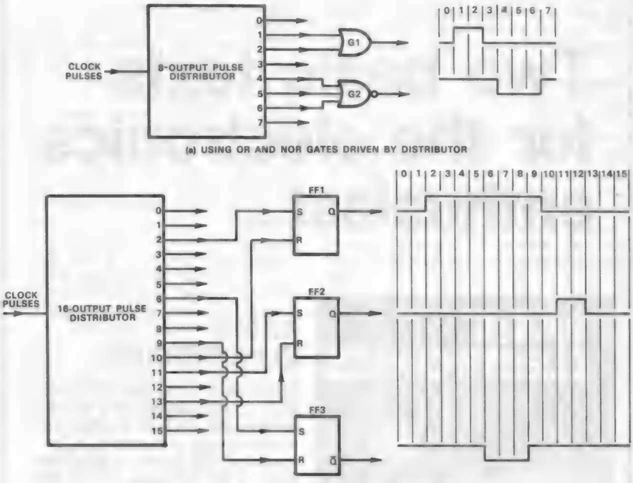
When this must be done, a circuit known as a synchroniser is used to perform synchronous gating in response to the non-synchronous control signal. Two commonly used synchroniser circuits are shown in Fig. 2.

As you can see from the basic synchroniser circuit shown in (a), there is only one other element apart from the AND gate used for the actual pulse gating. This is a D-type flipflop, with its Q output fed to the control input of the gate. As well as being fed to the gate, the clock pulses are also fed to the CLK input of the flipflop, while the non-synchronous control signal is fed to the D input.

Because the gate is controlled by the flipflop, it can only open and close when the flipflop changes state. And as the CLK input of the flipflop is driven by the clock pulses, it can only change state in a synchronous fashion—usually in response to either the positive-going or negative-going edges of the clock pulses, depending upon its design.

All the control signal does is "set the stage", as it were, providing the conditions for the flipflop to change state one way or the other. The actual changes of state take place under the control of the clock pulses.

This may be seen from the waveforms



(a) USING OR AND NOR GATES DRIVEN BY DISTRIBUTOR

(b) USING A DISTRIBUTOR TO DRIVE RS FLIPFLOPS

FIG. 4 : TIMING SIGNAL GENERATION

shown. While the control signal remains at the logic 0 level, the flipflop remains in its reset state and the gate remains closed. Then when the control signal changes to the logic 1 level (randomly), nothing happens immediately. However, if it is still at the 1 level when the next clock pulse arrives, the FF will be switched to the set state—and synchronously. The diagrams assume that the actual change of state takes place on the negative-going edge of the clock pulse, so that the pulse which performs the switching does not itself pass through the gate. The gate will open just after it has ended, due to propagation delays.

By the time the next clock pulse arrives, however, the AND gate will be fully open, so that this pulse will pass through. And following clock pulses will be passed also, until the control signal falls to the logic 0 level.

When this occurs, the next clock pulse to arrive will pass through the gate, but at the same time it will switch the flipflop to its reset state and hence "close the gate behind itself". By the time another clock pulse arrives, the gate will have closed and no further clock pulses will be passed until a new sequence is initiated by the control signal.

This type of basic synchroniser may be used whenever clock pulses or other timing signals must be gated synchronously in response to a single random input. However, there are occasions where two separate random signals must be used to control the gating—one to open the gate, and the other to close it. For applications of this type the modified synchroniser circuit of Fig. 2(b) may be used.

As you can see, it differs from the basic synchroniser only by the addition of an R-S flipflop. The S and R inputs of this element are controlled by the two asynchronous control signals, labelled here "start" and "stop". In turn the Q output of the R-S flipflop is used to control the D input of the gate control flipflop FF1.

In effect, the additional flipflop serves to combine the two asynchronous control signals into a single signal, fed to the D input of FF1. Apart from this the circuit operates in exactly the same manner as that of Fig.2(a).

Another requirement in many digital systems is for various operations to be performed in a sequence—A, then B, then C, and so on.

To arrange such a sequence of operations, it is often necessary to generate a series of single timing pulses or gating signals which are staggered in time. Circuits used to generate such a series of staggered pulses are generally called pulse distributors, and two typical pulse distributor circuits are shown in Fig.3.

The type shown in (a) consists of a synchronous binary counter driving a decoder. As the counter cycles through its counting sequence, the outputs of the decoder go to the active level in sequence also, producing the desired series of staggered pulses. Each pulse will be one clock period long.

Note that the counter used must be of the synchronous type, with all of its outputs changing state simultaneously. Otherwise the decoder will produce short spurious pulses or "glitches", as the counter outputs pass through transient counts (as we saw in chapter 10).

Although the circuit of Fig. 3(a) is shown with a 3-bit counter and decoder, producing a sequence of 8 staggered pulses, the same idea can be used to generate both shorter and longer sequences. Thus a 2-bit counter and decoder will produce a 4-pulse sequence, while a 4-bit counter and decoder will produce a 16-pulse sequence.

Similarly although the decoder is shown here having active-high outputs, producing active-high pulses, this is again not fixed. If active-low pulses are required, it is merely necessary to use a decoder with active-low outputs.

The second type of pulse distributor shown in Fig.3(b) is based on a shift register, with a number of stages equal to the number of staggered pulses required.

This type of circuit operates by first loading a "1" into the first stage of the register, along with "0's" into the remaining stages. Then the 1 is shifted along the register by clock pulses, so that the desired sequence of staggered pulses is produced by the Q outputs of the register flipflops.

The register shown is an 8-stage element, connected to produce an 8-pulse sequence. As you can see parallel input P0 is connected to logic 1 level, while the remaining parallel inputs connect to logic 0 level. The parallel load input PL is driven by an AND gate, which in turn is connected so that when Q7 of the register is at logic 1, the register is loaded from the parallel inputs.

Again if the distributor is required to have active-low outputs instead of active-high outputs as shown, this can be achieved by loading a single "0" into the register. To do this P0 would be taken to logic 0, and the remaining parallel inputs to logic 1.

As before, this type of distributor may be designed to produce staggered pulse sequences of any desired length. In this case it is merely a matter of providing the appropriate number of stages in the shift register.

Like the counter-type distributor, this type produces output pulses whose length is equal to one clock pulse period.

One of the most important applications of pulse distributors is for generating timing signals—gating signals which must last for a specific number of clock pulse periods, with a certain time relationship to other signals.

There are again two broad ways in which such signals may be generated, using the outputs of a pulse distributor. The two methods are illustrated in Fig.4.

As you can see, the method shown in (a) involves the use of OR and NOR gates, with their inputs connected to appropriate outputs of the pulse distributor. The number of inputs required by a gate depends upon the number of clock periods which must be spanned by the timing signal to be generated.

Hence gate 1, a 2-input OR gate with its inputs connected to the 1 and 2 ouputs of the distributor, will produce an active-high timing signal which will be active for clock periods T1 and T2. Similarly gate 3 with its three inputs connected to distributor outputs 4,5 and 6 will generate a signal which will be active for clock periods T4, T5 and T6, although in this case the signal will be active-low since G2 is a NOR element.

One disadvantage of this method is that

a timing signal required to span a large number of clock periods can call for a gate having an inconvenient number of inputs.

This disadvantage is not present in the second method, illustrated in Fig.4(b). Here R-S flipflops are used to generate the timing signals. The S input of the FF is connected to a suitable distributor output to time the start of the required timing signal, while the R input is connected to a second and later output to time the end of the signal.

Hence FF1, with its S input connected to the distributor 2 output and its R input to the 10 output, generates a signal which is active for clock periods T2-T9 inclusive. Similarly FF2 with its inputs connected to distributor outputs 11 and 13 generates a signal which is active for periods T11 and T12, while FF3 with its inputs connected to distributor outputs 6 and 9 produces a signal active for periods T6-T8 inclusive.

Note that whereas the S input of each flipflop is connected to the distributor output corresponding to the first clock period of the required timing signal, the R input must be connected to the output corresponding to the next clock period after the last period in the required timing signal. This is because the R input triggers the end of the timing signal, not the start of the last clock period.

As with the gating method, both active-high and active-low timing signals may be generated as required. In this case it is merely a matter of taking the signals from either the Q output or Q-bar outputs of the flipflops. The Q output produces an active-high signal, as shown in FF1 and FF2, while the Q-bar output produces an active-low signal as shown in FF3.

A further requirement which crops up from time to time in many digital systems is for pulses to be delayed by a short time, where the time involved may not be equal to an integral number of clock pulses. Some of the common techniques used to produce this type of pulse delay are shown in Fig. 5.

The simplest method is shown in (a). Here a familiar R-C integrator circuit is used to generate the delay, with a Schmitt trigger element following it to restore the pulse to rectangular form. The delay time is a function of both the product of R and C (i.e., the time-constant), and the thresholds of the Schmitt element.

This type of delay circuit is usually only used for relatively short delays, as the Schmitt element hysteresis tends to change the length of the pulse significantly when long delays are used.

Note that in the simple form shown, this circuit inverts the pulse. If the inversion is not required, a further inverter must be used to restore the original polarity.

The other main method of achieving a pulse delay is by using a monostable multivibrator, also called a "one-shot" Elements of this type are available in most logic families; they operate as shown in Fig.5(b).

As you can see, the device has two complementary outputs, like a flipflop. Generally there are two trigger inputs provided, one sensitive to a positive-going signal edge, and the other to a negative-going edge.

Basically the monostable has one stable state, corresponding to the "reset" state of a flipflop: with the Q output at logic 0 and the Q-bar output at logic 1. When triggered by an input signal, it switches from this state to the opposite state, but stays in the latter state only for a time determined by the external R and C time-constant. It then switches back automatically to its stable reset state.

Fairly obviously, a single monostable like that shown in Fig.5(b) is only capable of generating a time delay ending in a level transition. By itself it cannot generate a delayed pulse. If this is required, two monostables must generally be used, connected as shown in Fig.5(c).

Here the first monostable is used to generate the actual delay, with a time determined by R1 and C1. Then by connecting the Q-bar output of this element to the positive-going trigger input of the second element, the end of the delay time is made to trigger the second monostable and generate an output pulse whose length is determined by R2 and C2. So that the first monostable produces the actual delay, while the second generates the new pulse.

By the way, note that neither of the circuits of Fig.5(a) and (c) actually delay the original pulse. Both generate a second pulse, delayed from the input pulse, whose length is not necessarily the same as that of the original.

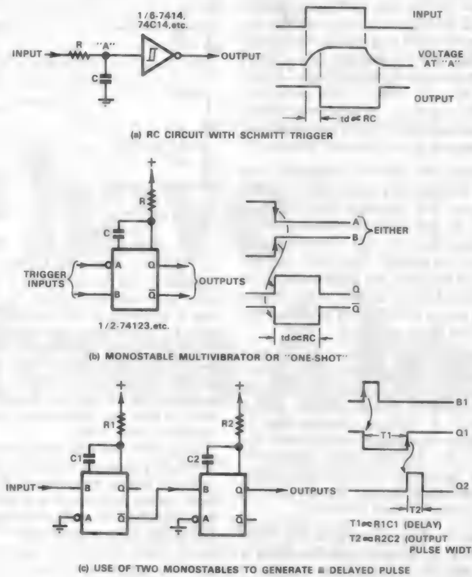
Before we close this chapter, there are two relatively minor aspects of control signal generation which should perhaps be mentioned.

One is the matter of deriving logic signals from mechanical switches, such as those of pushbuttons, keyboard switches, relay contacts and so on.

Generally speaking, this superficially simple operation tends to be complicated because most mechanical switches exhibit a characteristic known as "bounce". Instead of opening and closing cleanly, the switch contacts tend to vibrate rapidly both when they are meeting and when they are parting. This does not cause problems in most analog circuits, but in digital circuits it can produce all sorts of malfunction unless the "bounce" is suppressed in some way.

One approach is not to use mechanical switches at all, replacing them with equivalent switches which do not exhibit bounce. Examples of the latter are mercury switches, where the switching is performed by liquid mercury metal meeting and parting, and Hall-effect switches where the switching takes place in a solid-state element when a permanent magnet is brought near it or taken away.

While this approach is becoming more practical as time goes on, bounceless switches still tend to be rather more costly than the simple mechanical type. As a result it is often more attractive to use the latter, and use circuits such as those shown in Fig. 6 to "debounce" their output.

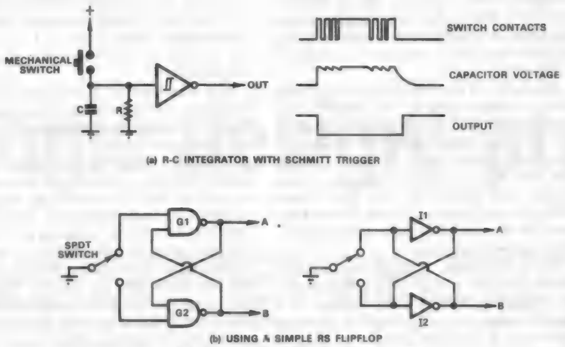


(a) RC CIRCUIT WITH SCHMITT TRIGGER

(b) MONOSTABLE MULTIVIBRATOR OR "ONE-SHOT"

(c) USE OF TWO MONOSTABLES TO GENERATE A DELAYED PULSE

FIG. 5 : DELAY CIRCUITS



(a) R-C INTEGRATOR WITH SCHMITT TRIGGER

(b) USING A SIMPLE RS FLIPFLOP

FIG. 6 : DEBOUNCING MECHANICAL SWITCHES

The method shown in (a) uses a capacitor C and resistor R to integrate the contact bounce. When the switch is closed, capacitor C charges up very rapidly as soon as the contacts first touch, taking the input of the Schmitt element well above its positive-going threshold. The R-C timeconstant is such that if the contacts bounce, the capacitor voltage only drops slightly, not enough to fall to the negative-going threshold of the Schmitt element.

Then when the switch is released, the capacitor voltage only drops significantly when the parting contacts have finished bouncing. By the time the voltage drops to the negative-going Schmitt threshold, the bounce has ceased. Thus both turn-on and turn-off bounces are suppressed. The R-C timeconstant is usually made about 20ms, as this is sufficient to suppress the bounce from most switches.

One disadvantage of this circuit is that it introduces a delay when the switch opens. This can cause problems where the switch is used to convey critical timing information. A further disadvantage is that two discrete passive components are required—the resistor and capacitor. These are becoming increasingly costly, both as components and in terms of the labour involved in their assembly.

Because of these problems the more popular way of performing debouncing is by using simple R-S flipflops, as shown in Fig.6(b). Here the flipflop effectively integrates the bounce, while at the same time introducing no delay for either closing or opening of the switch. The flipflop may be made up as shown using either a pair of gates or a pair of inverters—or even one of each. Often there are a couple of elements which happen to be "spare" in some of the ICs already used in the system, so that the debouncing is done at virtually no extra cost.

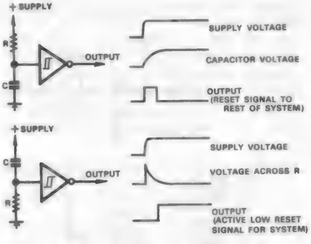


FIG. 7 : POWER-ON RESET CIRCUITS

The final matter we should discuss briefly is power-on reset circuits.

Many digital systems must be "initialised" when power is first applied to them, to ensure that all parts of the system begin operating from a predictable and known state. This initialisation is generally performed by generating what is known as a "power-on reset signal", which is fed throughout the system to force the various circuits and elements into known states.

The power-on reset signal may be generated by circuits of the type illustrated in Fig.7. Both circuits shown use an R-C charging circuit together with a Schmitt trigger element, as you can see.

In the upper circuit, the Schmitt element senses the capacitor voltage, so that when power is first applied the Schmitt element input is held below its threshold while the capacitor charges via R. The output of the element thus goes high. However, once the capacitor charges to the point where its voltage exceeds the Schmitt threshold, the element switches and its output remains low until power is removed from the system. Hence this circuit generates an active-high reset pulse whose length is proportional to the R-C timeconstant.

Where an active-low reset pulse is required by various parts of a system, this can be derived from the active-high signal using one or more inverters. However, if an active-low reset signal is required by all parts of the system, an easier approach is to use the second circuit of Fig.7. Here the R and C are reversed, so that the Schmitt element senses the voltage across R.

As a result, when power is first applied the input of the Schmitt element immediately rises above its threshold, and remains there while the capacitor charges. The output of the Schmitt element thus remains low at switch-on, to generate the reset signal. However, as the capacitor charges the voltage across R decays exponentially, and eventually reaches the negative-going threshold of the Schmitt element. The element then switches, its output going high and remaining in this state until power is removed.

Actually it is not essential to have the Schmitt element in either of the circuits of Fig.7, as neither depends upon hysteresis. A normal inverter element may be used if desired, although the Schmitt element tends to produce a more rectangular output.

In cases where a power-on reset signal is only required for a single logic element, it may be possible to omit an inverter altogether, and simply connect an R-C circuit to the reset pin of the element itself.

Chapter 17

Memory: RAMs

Memory devices form an important part of many digital systems, the most obvious example being in computers. This chapter discusses random-access memories or RAMs, in particular semiconductor RAMs. It deals with both bipolar and MOS devices, static and dynamic RAMs, non-volatile RAMs and specialised devices like the FIFO and the content-addressable memory.

In many digital systems there is a need to store significant amounts of information, usually in the form of multi-bit binary numbers. A good example of this is in computers, where a sequence of numbers must be stored to provide the computer with its "program". It is often also necessary to store the actual data which the computer is to process, and the results of calculations.

Another example of the need for storage is in video display terminals, where a storage "buffer" must be used to hold the information to be displayed, so that the video scanning circuitry can continuously refresh the screen. A similar need exists in many mechanical printers, where it is often necessary to store incoming information temporarily, until the printing mechanism can deal with it.

Still another need for storage arises in systems which must perform code conversion, or generate numbers related to others by relatively complex mathematical functions. In many such applications it is easier and more efficient to have a stored "lookup table" of numbers, than to try calculating them. The table of stored numbers is used in a very similar way to the tables of logarithms or trigonometric functions which were at the elbows of mathematicians and students, before the era of pocket calculators!

Organised storage or "memory" facilities thus form an important part of many digital systems. They come in many different forms, each of which tends to offer a particular combination of such characteristics as cost, storage capacity, and the ease and speed with which information can be stored and retrieved.

The various types of memory system and device tend to fall into two broad groups. There are those which store information in serial fashion, so that in order to gain access to a particular storage location or "cell", it is necessary to perform some sort of serial scanning operation. These types of memory are known as serial-access memories.

Examples of serial-access memories are those using magnetic tapes or discs, and those using new technologies such as charge-coupled devices (CCDs) and magnetic "bubble" devices.

Historically serial-access memories were the first to be developed, in the form of punched paper tape and cards, mercury delay lines and magnetic drums. They still tend to be used for very high capacity or "bulk" storage, and for storage of information which does not have to be accessed frequently, because they tend to offer the lowest cost per unit of storage.

Serial-access memories will be discussed in more detail in a later chapter.

The other main group of memory devices and systems are those which store information effectively in parallel, so that any particular storage location or cell is just as accessible as any other, and can be reached directly. These types of memory are known as random-access memories, or "RAMs".

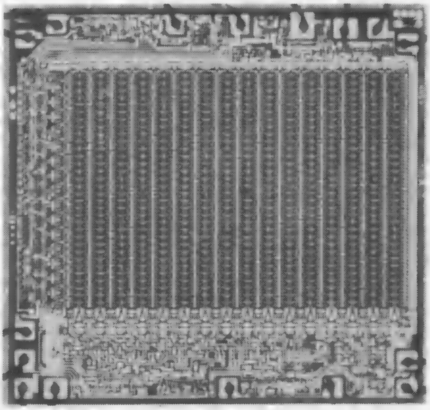
A useful way of visualising random-access memories is to think of them as an array of "pigeon-hole" cells, rather like the letter rack which used to be behind the reception desk in hotels. Information may be stored or "written" into any cell at random, with equal ease, and retrieved or "read" just as easily.

The first type of random-access memory to be used extensively was the magnetic core memory. This consisted of an array of tiny toroid cores or "doughnuts", moulded from magnetic ferrite material. Each core could be magnetised in one direction or the other, and so used to store one bit of information. Sets of wires threaded through the cores were used to write information into them, and also to read it from them.

Magnetic core memories of this type were capable of quite rapid access: in the order of 1 microsecond. Like most serial-access memories they were also non-volatile, in that the stored information does not "evaporate" when power is removed from the memory. However they were also relatively costly, as the arrays of cores had to be threaded on the wires almost entirely by hand.

Other types of magnetic random-access memory were developed in an effort to reduce the cost, including memories in which thin films of magnetic ferrite material were plated onto the various magnetising and sensing wires. However, this approach to random-access memory has been almost completely superseded by integrated circuit memories, based on various types of semiconductor technology. One of the main factors behind the growth of IC memories has been their very much lower cost per stored bit of information, together with the dramatic reduction in memory size.

There are quite a number of different types of random-access IC memory, but most of them fit into two broad categories.



This is an enlarged view of a 1024-bit static RAM device using bipolar technology. The actual chip is less than 2.5mm square. The array of storage cells is clearly visible.

One of these includes those memories which permit information to be written into storage cells with as much ease as it may be read out, so that they may be used in any situation requiring frequent and random writing and reading. Memories of this type should strictly be called "read/write random-access memories" or "R/W RAMs", but it has become common to refer to them simply as "random-access memories" or RAMs.

The other main type of IC random-access memories includes those devices which either do not permit repeated writing of information into the storage cells, or do not allow repeated writing with the same ease that they allow repeated reading. Compared with the first type of memories, these devices are more suitable for use in applications where information needs to be written in either only once, or infrequently, most of the time being read out. As a result, memories of this type are given the general title of "read-only memories", or ROMs.

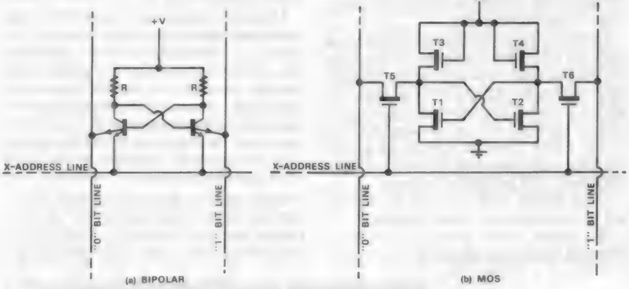


FIG. 1 : STATIC RAM MEMORY CELLS

There are various types of ROM, some of which have the stored information effectively written into them permanently during manufacture, and others designed so that they may be written into or "programmed" by the user, either permanently or semi-permanently. The latter types are generally known as programmable ROMs, or "PROMS".

ROMs and PROMs are very important devices, and deserve a discussion in their own right. For this reason, let us leave them now until the next chapter, and spend the remainder of the present chapter looking more closely at the first type of IC memories: RAMs.

RAMs are divided into two different types. In one type, the information is stored in what is basically an array of many flipflop latches, each storing one bit of information, and storing it stably until either the latch is reset, or the power is turned off from the RAM as a whole. RAMs of this type are known as STATIC RAMs.

In contrast with this approach, the other types of RAM store their information in an inherently less stable fashion—typically as charges in an array of capacitors. Each capacitor stores a charge corresponding to a single bit of information, but unlike the latches of a static RAM the capacitors tend to lose the stored information due to charge leakage. To retain the stored information, the charges therefore have to be sensed and "refreshed" repetitively. These types of memory are thus known as DYNAMIC RAMs, to indicate that they need attention on a continual or dynamic basis.

Looking first at static RAM devices, these are again divided into two broad types: those using bipolar technology, and those using MOS technology. Fig. 1 shows the basic memory cells used in each type, and as you can see they are both essentially simple latch flipflops.

The bipolar cell shown in (a) uses only two transistors, as you can see, each with a collector load resistor R. The base of each is connected to the collector of the other, so that whichever is conducting holds the other off and vice-versa. The resistors R are generally formed by diffusion from the same silicon as the transistors, with a value as high as possible in order to make the current drain and power dissipation of the cell extremely low.

The MOS cell shown in (b) is very similar, with the lower pair of transistors T1 and T2 in the same cross-coupled configuration. However, in place of the load resistors there is usually a further pair of MOS transistors as shown (T3 and T4), connected as constant-current devices. As with the bipolar cell, these are designed to minimise the cell's current drain and power dissipation, so that it stores information as efficiently as possible.

In place of the constant-current load transistors T3 and T4, some recent static MOS RAM cells use very high value ion-implanted polysilicon resistors. These have a value as high as 5000 megohms, bringing the average dissipation of a storage cell down to around 20 microwatts.

Basically a static RAM consists of a large number of cells like those in Fig. 1 (a) or (b), connected in a two-dimensional array or matrix. Each cell in the array is accessible by means of "address lines" in the two main axes of the array, usually designated the "X" and "Y" axes for convenience, or sometimes the "word" and "bit" axes respectively.

As you can see, the bipolar memory cell of Fig. 1(a) is connected to an X-address line, which in fact forms the main common emitter return path to the supply. A typical array will have quite a number of these X-address lines, each controlling a row of cells. The additional emitter of each transistor is individually connected to a separate Y or bit line, as shown. There will again be quite a number of pairs of these lines in a typical array, each pair associated with a column of cells, and used to write into and read from them.

The MOS memory cell of Fig. 1(b) is connected into its array in a similar manner, although here the X-address line does not control the storage latch itself, but two transistor switches T5 and T6 which connect the latch collectors to the vertical bit lines.

In each case, the basic idea is that the X-address lines of the array are used to select a particular row of memory cells, while the pairs of bit lines are then used to write into or read from the various cells in the selected row.

With the bipolar RAM cell, the X-address lines are normally held near the negative supply rail. The bit lines are held at a slightly higher voltage, so that the output coupling emitters of all cells in the column are slightly reverse biased. Under these conditions, the cells are quiescent, and will store information as long as power is applied.

To write into a cell, its X-address line is raised to within about a volt of the positive supply rail. While it is high, either the 0 or 1 bit line is also taken high, depending upon the data bit to be written into the cell. Then the X-address line is allowed to return to its quiescent level, followed by the bit line.

As the address line falls, the output coupling emitter connected to the high bit line is reverse biased, whereas that on the other side of the cell is forward biased and able to conduct. The cell is therefore "steered" in the appropriate direction, aided by its own inherent regeneration. By the time the address line reaches its quiescent level, allowing the two coupled emitters to assume control, the cell has stabilised with its new stored bit value; the transistor whose coupling emitter was taken high will be cut off, while the other transistor will be conducting.

To read from a cell, only the X-address line is taken high, while the bit lines are allowed to "float" and their potential sensed. When the address line rises, the coupled emitters become reverse biased, and the current of the conducting transistor transfers to its output coupling emitter. The voltage on that bit line therefore rises, indicating the value of the bit stored in the cell

The readout is non-destructive, as the cell returns undisturbed to its quiescent state when the address line is returned to its low level.

To write into the MOS cell of Fig. 1(b), the X-address line is first taken high to turn on the two switch transistors T5 and T6. Then one of the two bit lines is taken briefly to the negative supply rail, depending upon the value of the bit to be stored. This drags down the corresponding side of the cell, forcing it to switch to the desired state (or to remain in that state if it is already there). The address line is then returned to its normal low level, turning off T5 and T6, and isolating the cell once more.

Reading the MOS cell is done in much the same way as with the bipolar cell. The address line is taken high, turning on T5 and T6 and allowing the "off" or high side of the cell to raise the potential of the corresponding bit line. The bit line potentials can thus be sensed to discover the value of the stored bit.

Naturally enough, because all of the cells in a particular row of the memory share ■ common X-address line, they are all potentially read whenever any cell in the row is either written into or read from. This happens with both the bipolar and MOS cells, but it causes no problems because further logic is used to select the desired bit line pair (or pairs, if ■ number of cells are being accessed at the same time). As readout is non-destructive, the unselected cells in the same row are only temporarily disturbed.

For convenience, the cells in a typical RAM are generally arranged in a roughly square array, of a number of such arrays in the case of a large RAM. Each array has approximately the same number of X-address lines and pairs of bit lines, so that a RAM with say 1024 cells would very likely have them arranged in a 32x32 array—with 32 X-address lines and 32 pairs of bit lines.

This internal geometric arrangement within a RAM is quite distinct from the way it may be designed to appear "organised" from the fuctional point of view, as seen by external circuitry. For example the same 1024-bit RAM may be arranged so that only one cell is selected at any one time, in which case it is said to be organised as a 1024 x 1 RAM. On the other hand, if it is arranged so that say four cells are selected together and written and read in parallel, it would then be said to be organised as a "256 x 4" RAM.

It could even be arranged so that eight cells are selected together, thus having an organisation of "128 x 8".

Fig. 2 shows how this sort of organisation is achieved. It shows a 256-bit RAM, with the storage cells arranged physically in a 16 x 16 array. However, it is functionally arranged so that four cells are selected at a time—i.e., it is organised as a 64 x 4 RAM.

Functionally it therefore has 64 effective storage "addresses", each of which can store 4 bits of information. To select these 64 addresses, the RAM must be provided with a 6-bit address code (6 bits have ■ total of 64 truth-value combinations). The address is applied to the six inputs on the lower left of the diagram. The four data bits read out from the address specified appear at the four data terminals at the lower right, while the same four terminals are used as inputs if data is to be written into the address.

As you can see, four of the address inputs are taken to a decoder, whose 16 outputs are fed through drivers to control the X-address lines of the storage array. The remaining two address inputs are fed to a second decoder whose four inputs are used by the driving and sensing circuitry associated with the array's bit lines. In this case, each output line of the Y-decoder would be used to select a group of four bit-line pairs, effectively connecting them to the data terminals.

There is also ■ timing and control section within the RAM, as shown, whose function is to co-ordinate read and write operations. This section is provided with a logic input, so that external circuits can direct whether a write or read operation is to be performed. The label "READ/WRITE-BAR" indicates that a logic high corresponds to read, while a logic low corresponds to write.

The same section of the RAM is also often provided with one or more "chip select" logic inputs, as shown. These are to allow overall access to the RAM to be controlled conveniently—a feature which becomes very useful where a number of such RAMs are to be used together, to form a large memory. Each RAM's chip enable input can be controlled by one of the higher-order address bits, or from an output of a decoder driven by the higher address bits.

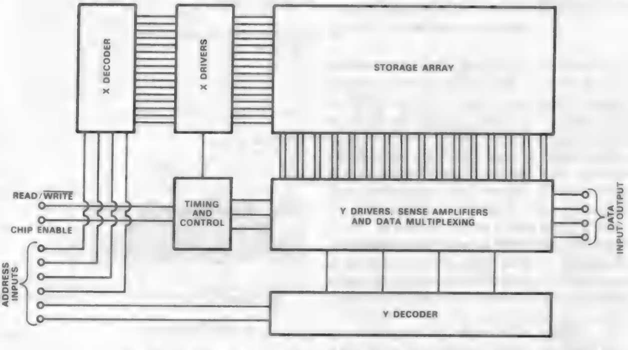


FIG. 2 : BASIC READ/WRITE OR "RAM" MEMORY ORGANISATION

Static RAMs of the type we have considered so far are very widely used in modern digital systems. They are easy to use, and provide fairly fast memory facilities at a moderate cost—far less than that of core memories.

At the time of writing, the largest static RAMs being made have a capacity of 4096 bits, and both bipolar and MOS devices are made with this capacity. The bipolar devices tend to be faster—i.e., to offer a shorter access time—but they tend to have higher power consumption. Typical bipolar devices have an access time of around 100ns (nanoseconds), while MOS devices have an access time of around 220ns.

Where high memory capacity is required, both of these main types of static RAM tend to have disadvantages. One is that their power consumption is still quite significant, as power is continuously dissipated by the storage cells.

If low power consumption is essential, an alternative type of static RAM tends to be used. The most common such device uses CMOS (complementary MOS) technology, with a silicon-on-sapphire construction. Although it is about five times more costly than a normal NMOS static RAM, the power consumption is very much less for the same capacity and access time. For example a 1024-bit/150ns device typically dissipates only 4mW (milliwatts), compared with around 300mW for a comparable NMOS device.

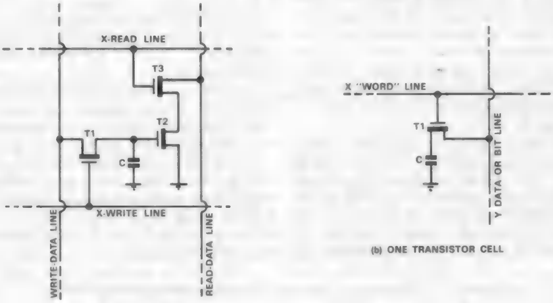
The other main disadvantage of static RAMs where large memories are concerned is cost. Even for the lower cost MOS type, the cost at the time of writing (mid 1977) is around 0.25 cents per bit. This may not seem high, but it soon adds up where large memories are concerned. As a result, large memories tend to be made using not static RAMs, but the dynamic type mentioned earlier.

In terms of basic organisation and function, dynamic RAMs are not greatly different from the static type we have discussed so far. The main difference lies in the actual memory cells themselves. As mentioned earlier, instead of flipflop latches like those of Fig.1, the cells of dynamic RAMs are based on a capacitor. Two fairly common types of dynamic RAM cell are shown in Fig.3.

As you can see the cell in (a) uses three MOS transistors together with the storage capacitor, shown as "C". Actually the capacitor is generally not a separate physical component, but in fact the gate-channel capacitance of transistor T2. Although only a few hundredths of a picofarad, it can store charge for around 2ms because of the high leakage resistance present.

With this cell there are two X-address lines as well as two data bit lines. One X-address line and data bit line are used for writing, and the other X-address line and bit line for reading. To write data into the cell, the X-write line is taken to the high level to turn on switch transistor T1. At the same time the write data bit line is taken high or low, depending upon the data bit to be stored. Capacitor C thus acquires the appropriate charge, which remains when the X-write line is returned to its low quiescent level.

To read data from the cell, the X-read line is taken high. This turns on switch transistor T3, allowing the read data line to sense the conducting state of transistor T2. If this is an enhancement-mode device, it will be a low impedance if C is charged, or a high impedance if C is uncharged.



(a) THREE TRANSISTOR CELL

(b) ONE TRANSISTOR CELL

FIG. 3 : DYNAMIC RAM MEMORY CELLS

The cell of Fig. 3(b) is somewhat simpler than that just described, using a single MOS switching transistor and requiring only a single X-addressing or "word" line and a single Y-data or "bit" line. The capacitor C is separate from the transistor, and is typically formed by a small area of polysilicon material above a diffused bit line, with a thin layer of silicon dioxide as dielectric.

Operation of this cell is very similar to that just described, except that both read and write data are transferred via T1 to the single bit line.

Actually because the stored charge tends to leak away in the cell capacitors, dynamic RAMs are usually arranged so that every read operation is automatically followed by a write operation, to restore the charge. This is done by the sensing and driver circuitry connected to the bit lines, so that all of the cells in a particular X row are automatically refreshed whenever any cell in the row is either written into, or read from.

As the effective storage time of typical cells is around 2ms, all cells in a dynamic RAM must be refreshed at least every 2ms. Special circuitry is usually required to perform this refresh servicing, on an automatic basis. The circuitry effectively "commandeers" the dynamic RAM every so often, and refreshes it by performing read operations. However this doesn't take long, as it is only necessary to perform as many read operations as there are X-address lines in the RAM's cell array. The internal circuitry automatically ensures that all row cells are refreshed.

At the time of writing, dynamic MOS RAMs are made with capacities up to 16,384 bits, and offering a cost of .08 cents per bit. As you can see, this is considerably lower than the figure for static RAMs. The access time is around 250ns, comparable with NMOS static RAMs.

It is predicted that dynamic RAMs with capacities of 65,536 bits will be in production by 1979 with a significantly lower cost again.

One of the disadvantages shared by all normal semiconductor RAMs, whether static or dynamic, is that they provide only "volatile" storage. The storage is dependent upon power being applied, so that if power is removed from the RAM, its stored information is lost.

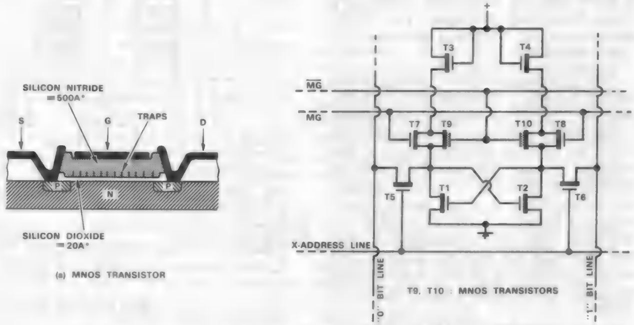
Not surprisingly, there are situations where this volatility can be inconvenient to say the least. As a result, a number of alternative memory devices have been developed to provide non-volatile storage. These include the ROM and PROM devices to be discussed in the next chapter, bubble memories and magnetic discs and tapes (also to be discussed, in later chapters).

Just recently, non-volatile semiconductor static RAMs have begun to appear, and these seem likely to provide a further useful alternative to standard devices.

One device of this type has been developed by the Tokyo Shibaura company in Japan. It uses metal-nitride-oxide-semiconductor (MNOS) transistors to provide the non-voltatile storage. These do not take part in the normal write-read operation of the RAM cells, however, being used only to retain the stored information when power is removed.

The structure of an MNOS transistor is shown in Fig. 4(a). As your can see, it is not too different from a normal MOS transistor. The main difference is that there are two dielectric layers between the semiconductor channel and the insulated metal gate (G). Only one layer is silicon dioxide, and this is very thin—approximately 20 angstroms (2 nanometres). Above this is a layer of silicon nitride which is somewhat thicker, around 500 angstroms (50nm).

The idea of the two dielectric layers is that where the two layers join, crystalline defects are formed which can act as "traps". Under certain conditions, these traps can capture carriers. The trapped carriers effectively form a charged second gate of the transistor, and can control the transistor's channel conductivity independently of the main gate. And because of the high resistivity of the oxide and nitride layers, the trapped charge leaks away extremely slowly. Typically it can be retained for more than a year, which is long enough to be regarded as non-volatile for most purposes.



(a) MNOS TRANSISTOR

T9, T10 : MNOS TRANSISTORS

(b) MEMORY CELL

FIG. 4 : NON-VOLATILE MNOS RAM

Fig. 4(b) shows how a pair of MNOS transistors are incorporated into what is basically a standard MOS RAM cell. If you compare the diagram with that of Fig. 1(b), you will see that it is basically the same cell with four additional transistors. T7 and T8 are two normal MOS transistors, while T9 and T10 are two MNOS devices.

In normal operation, the logic levels on the additional gating lines MG and MG-bar are arranged so that T7 and T8 are continuously conducting, with T9 and T10 continuously held off by their metal gates. The cell is thus able to operate as a completely normal MOS memory latch, and can be written into and read from exactly as described earlier. The MNOS transistors play no part in this normal operation.

However when power is to be removed, normal operation of the RAM is stopped. The logic levels on the MG and MG-bar lines are then reversed, switching off T7 and T8 and switching on T9 and T10. This causes the trap layers of these transistors to acquire charges, the size of the charges mirroring the current state of the latch—and hence effectively capturing the stored bit. The power may then be removed, leaving the stored charges trapped in the MNOS devices.

When the power is eventually re-applied, the logic levels on the MG and MG-bar lines are arranged so that T9 and T10 are simultaneously turned on, while T7 and T8 are held off. This allows the charges stored in the MNOS devices to "steer" the latch as it turns on, so that the saved bit of information is effectively fed back into the latch. Then the logic levels on the MG and MG-bar lines are reversed once more, changing the cells back into standard RAM cells.

In effect, the MNOS static RAM is really two separate memories integrated together: a standard MOS RAM and a non-volatile memory using charge storage, with the ability to pass stored information between the cells of each.

Before we end this chapter, there are two further memory devices which should perhaps be mentioned. One is the "first-in-first-out buffer", or "FIFO", which at first sight may seem quite different from the devices we have been talking about. Yet it is basically a static RAM, in disguised and augmented form.

From a functional point of view, a FIFO appears like a number of parallel shift registers, but with a difference: information can be fed in at one end and out at the other, at different rates. In effect, information fed in at the "input" end of the FIFO appears to "trickle" down to the other, whereupon it can be taken from the "output" end at a different rate altogether, without losing its sequence.

Such a device can be very handy where information must be fed from a system working at one rate, into another working at a different rate.

Although the FIFO seems rather like a strange sort of "flexible" shift register, it is in fact a static RAM provided with some additional circuitry. The main additions are a pair of address counters, multiplexed so that either can control the RAM addressing circuits.

One counter is used to control the address into which the FIFO's data input is written; the other controls the address from which the outut data is read. The counters are automatically incremented each time a write or read operation is performed, respectively. Thus what happens inside the FIFO is that input data is stored in consecutive RAM locations at one rate, and independently read from those locations at a different rate.

There is additional circuitry which compares the contents of the two address counters, and provides two "housekeeping" logic signals: one to indicate when the FIFO is "full", and cannot currently accept any further input, and the other to indicate when the FIFO is "empty" and cannot currently deliver any output data. There may also be circuitry to allow both counters to be reset, so that it may be effectively "emptied" when power is first applied.

The other memory device which should be mentioned here is the so-called "content-addressable" memory, or CAM, also called the "associative" memory.

There are a number of different types of CAM, but from a functional point of view they all tend to have one thing in common. Stored data is not identified in terms of its absolute address in the memory, but by some characteristic of the data itself.

Writing into a CAM may be either a random operation, where new data is simply stored in any location not currently occupied, or it may be arranged so that data with certain common aspects—say some bits of a particular value—are stored together.

Reading from a CAM is done not by specifying a particular address, as with a RAM, but by specifying a data "key"—some necessary characteristic of the data that is sought, such as a certain bit pattern. The CAM then produces any of its contents which meet the criterion.

One type of CAM is basically a static RAM with an internal address counter, a clock, and a comparator which can examine memory contents and compare them with the "key" data supplied by the external circuit. When data is to be written into the CAM, the address counter is incremented until the comparator indicates an empty address. To read data out, the address counter is incremented until the comparator indicates a match between the stored data and the "key". The stored data thus found is presented at a set of output data terminals.

The content-addressable memory is a rather esoteric device, but finds use in code conversion applications and in systems designed to perform "associative reasoning".

Chapter 18

ROMs & PROMs

Continuing the discussion of memory devices, this chapter looks at semiconductor memories which are used for permanent or semi-permanent "read-only" storage. It describes the three main types of device in present use: mask programmed ROMs, fusible-link programmable ROMs (PROMs), and reprogrammable PROMs of the electrical and ultra-violet erasable type.

In contrast with RAM devices, which allow information to be written into and read from them with equal ease, ROM devices are designed primarily for read-only operation. They are thus suitable for use in applications where information must be stored permanently and used or referred to from time to time, and also for applications where information must be stored semi-permanently.

An important application of ROM devices is for program storage in microcomputer controllers, where the microcomputer is "dedicated"—i.e., intended to perform the same set or sequence of tasks for a very long period. By having the sequence of instructions which make up the microcomputer's program stored permanently in a ROM, they are immediately accessible whenever required, and do not have to be fed in each time the controller is required to operate.

ROM memory devices also have a place in many general-purpose computer systems, because such systems tend to have many utility programs which are frequently and repetitively used—like loaders, text editors, assemblers, compilers, interpreters and so on. By having these programs stored in ROMs as "firmware", they can be called instantly whenever they are required, obviating the need to load in each time from paper tape, magnetic tape or disc.

There is increasing use of ROMs also within the actual processor or "CPU" section of computers, to store the processor's microprogram algorithms. These are the sequences of internal sub-operations which are used to carry out each of the various instructions in the processor's repertoire. By having a processor microprogrammed with its algorithms stored in a ROM, it becomes possible to change the instruction set relatively easily, for different applications.

ROMs also tend to be used quite widely in both computers and other digital systems for the storage of permanent or semi-permanent reference data—such as logarithm tables, tables of complex mathematical functions and code-conversion tables. The last of these areas is very broad, covering a variety of things from conversion between communication codes (e.g., Baudot or Murray code to ASCII code and vice-versa), to generation of alphanumeric character scanning-row patterns from ASCII code inputs, as required for a video display terminal or line printer.

Yet another application of ROM devices is in programmable logic arrays, as we saw in an earlier chapter, to effectively synthesise complex logic functions. Here the information stored in the ROM is equivalent to the truth table of the complex logic function concerned.

In short, then, ROMs form an important group of semiconductor memory devices, which tend to be found in many digital systems. A knowledge of the various types of ROM and their operation is thus very worthwhile for anyone working with, or seeking to work with such systems.

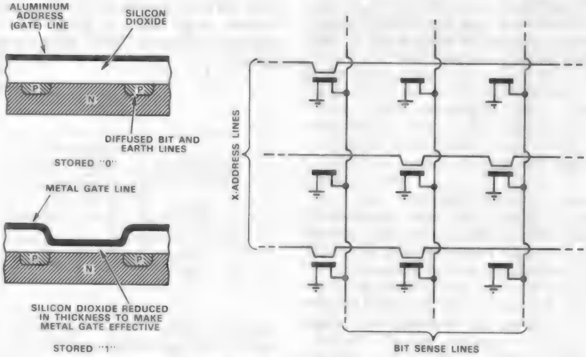

ALUMINIUM ADDRESS (GATE) LINE
SILICON DIOXIDE
P
N
P
DIFFUSED BIT AND EARTH LINES
STORED "0"
METAL GATE LINE
P
N
P
SILICON DIOXIDE REDUCED IN THICKNESS TO MAKE METAL GATE EFFECTIVE
STORED "1"
X-ADDRESS LINES
BIT SENSE LINES

FIG. 1 : MASK-PROGRAMMED ROM

Broadly speaking, there are three main types of ROM device in current use. One is the mask-programmed ROM, in which the stored information is effectively written in during manufacture. Then there is the fusible-link programmable ROM or "PROM", which is programmed permanently by the user. And finally there is the erasable PROM or "EPROM", which not only may be programmed by the user, but erased and reprogrammed many times.

Mask-programmed ROMs are so named because the information they contain is effectively stored in them during manufacture, from one of the photolithographic masks used in the fabrication process.

There are various types of mask-programmed ROM, which can be made using either bipolar or MOS technology. The former tend to be faster in operation, but also more expensive due to the larger number of fabrication steps involved in bipolar technology.

A mask-programmed ROM using MOS technology is illustrated in Fig. 1. Here the individual memory cells in the storage array are single enhancement-mode N-channel MOS transistors, with the P-type drain and source regions formed by diffusion. The diffusion regions are actually in the form of long strips, which also form the bit sense and earth lines for whole columns of cells in the array.

As with normal MOS transistors, the gates consist of a thin layer of aluminium vacuum deposited on the top of the silicon dioxide passivation. However, in this case the metallisation is in the form of thin strips running across the array, so that they also form the X-address row select lines.

The actual information is "stored" in the MOS transistor cells by controlling the thickness of the silicon dioxide passivation above the channel of each transistor, as shown. To effectively store a zero (0), the passivation is left at its full thickness. This makes the gate almost ineffective in inducing conduction in the transistor's channel, because of the separation between the two.

To effectively store a one (1), the passivation is etched to a reduced thickness above the channel. This allows the gate to become effective in inducing channel conduction, when the gate line is taken to the high logic level.

All of the cells in the ROM are programmed at the same time during device fabrication, by means of a photolithographic mask which controls etching of the passivation layer. Wherever cells are to be programmed with a "1", the mask has a tiny rectangular hole; thus the etching only takes place in those locations, leaving all other cells with full thickness passivation.

Mask-programmed ROMs made using bipolar technology generally have cells consisting of single NPN transistors, connected as potential emitter-followers. The bases of each row of transistors in the array are driven from the X-address lines, while the collectors are all connected to the positive supply rail.

The emitters of the transistors in each column of the array are potentially capable of being connected to Y-axis bit sense lines, formed by metallisation deposited on the silicon dioxide passivation. The ROM is programmed by using a custom mask to determine which of the emitters in each column have holes etched in the passivation above them, to allow the metal bit sense line to make contact with them.

In other words, the emitter of the transistor in each cell is only connected to the bit sense line for its column if that cell is to effectively store a 1. But if the cell is to store a 0, the emitter is simply left unconnected.

A ROM of this type is typically organised as shown in Fig.2. As you can see, it is very similar to that of a RAM as given in the last chapter. The main differences are that there is no read/write control line, and the data pins are purely outputs.

Note that Fig.2 shows a 1024-bit ROM, whose internal array is actually organised as 32 rows by 32 columns. The external organisation is different, however, being 128 x 8—i.e., the ROM is arranged so that it effectively stores 128 bytes each of 8 bits. There are thus seven address bit inputs, to define the 128 nominal memory addresses. Five of the address lines are decoded to produce the 32 X-address or row select lines for the array, while the other two address bits are decoded to select one of four 8-bit groups of column sense lines.

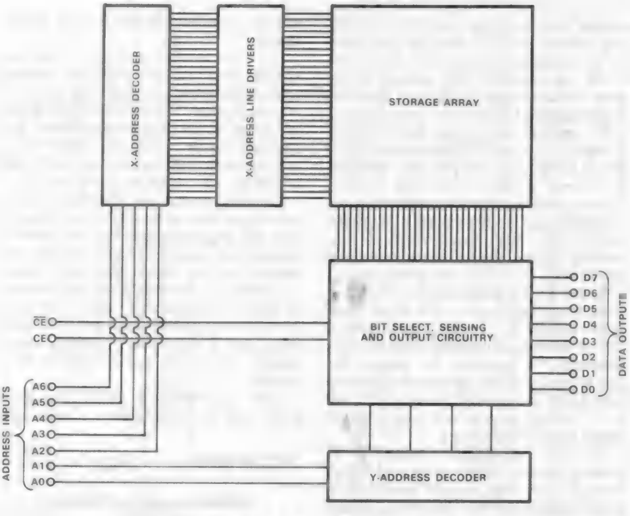


FIG. 2 : 1024-BIT ROM WITH TYPICAL ORGANISATION

Mask-programmed ROMs of the type we have been considering thus far can be made to store quite large amounts of information. At the time of writing, at least one manufacturer is marketing an N-channel MOS device with a capacity of 32768 bits (or "32k"), organised externally as 4096 bytes. It is likely that devices with double or quadruple this capacity will be available before 1980.

Like all of the photolithographic masks used to fabricate an integrated circuit, the mask used to program this type of ROM involves a very high degree of precision. This together with the setting-up required for the various steps in IC fabrication tends to make it uneconomical to produce mask-programmed ROMs in small quantities. But when a large number of identical ROMs are required, this type of ROM is generally far lower in cost than the other types we will consider shortly.

In short, the mask-programmed ROM is not very suitable for applications requiring a single custom-programmed ROM, or even those requiring a small number of identically programmed devices. And being permanently programmed, it is not really suitable for applications where the stored data must be changed from time to time. But where a large number of ROMs must be identically programmed on a permanent basis, this type of ROM turns out to be the most suitable and the lowest in unit cost.

In contrast with the way the mask-programmed ROM is programmed by the IC manufacturer during its fabrication, the two remaining types of ROM are programmable by the user. They are thus known as field-programmable ROMs or "PROMs". As you might imagine, the fact that they are user programmable makes PROM devices more suitable for applications requiring single or small quantities of custom-programmed ROMs.

Perhaps the simplest type of PROM to understand is the fusible-link device. As the name suggests, this is programmed by passing short pulses of relatively high current through small resistive links, so that they "fuse" or break. Generally fusible-link PROMs are made using bipolar technology, because bipolar devices are more suitable than MOS devices where high currents are involved.

The operation of a typical fusible-link PROM is illustrated in Fig. 3. As you can see, the cells in the storage array consist of single NPN transistors, connected as

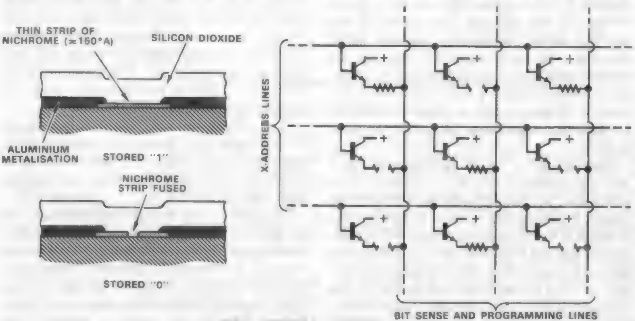


FIG. 3 : FUSIBLE-LINK PROM

emitter followers. The bases of the transistors in each row of the array are driven by the X-address lines, while the emitters of the transistors in each column of the array are connected (initially, at least) via small resistors to Y-axis bit sense lines.

The emitter resistors generally consist of either a thin film of nichrome alloy, as shown in Fig. 3, or alternatively a similarly thin layer of polycrystalline silicon. Typically they have a resistance of around 150 ohms. When the device is made, all resistors are intact and effective in connecting the emitter of their associated transistor to the correct bit sense line.

As the PROM is supplied to the user, then, all storage cells in the array effectively contain ■ "1" (or alternatively a "0", if the output circuitry contains logic inverters). To program the device, the user must fuse open the links on all those cells which are required to store a "0" (or a "1" if the outputs are inverting). All others links are left intact.

The links are fused individually, by passing current pulses of around 30mA through them. This current level is much larger than the normal operating current. The pulses are fed into the links by activating special current driver transistors connected to the Y-axis bit sense lines, within the PROM. These transistors are held cut off by the normal operating voltages within the PROM, and have no effect on normal bit line sensing.

To fuse ■ particular link, the normal address code for its byte is first applied to the PROM's address inputs. This activates the X-address row select line driving the base of the link's transistor, and also connects the link's Y-axis bit sense line to the corresponding data bit output of the PROM, via the sensing and programming circuitry. Then a programming pulse is forced into the data bit output, using it as an input.

The programming pulse is not positive-going, but negative-going. In fact it takes the data bit output negative with respect to the normal negative supply rail, by about 6 volts. This has the effect of disabling the normal sensing circuitry, and at the same time enabling ■ normally inactive current driver transistor. This transistor accordingly conducts, pulling the bit sense line negative by about 4.5 volts and thus forcing a current of about 30mA through the link until it fuses.

The procedure is fairly straightforward, and does not call for complex or elaborate equipment. A fusible-link PROM can in fact be programmed with a very simple manual programmer, with toggle switches to select device addresses and data bits, a press-button to apply programming pulses, and positive and negative power supplies. However, to program a large PROM in this way can be rather tedious and time consuming.

The other main drawback of manually programming ■ fusible-link PROM is that the programming is permanent, so that mistakes can't be undone. If you fuse the wrong link by mistake, it can't be rejoined, and the entire PROM may have to be discarded.

For this reason most fusible-link PROMs tend to be programmed automatically, under the control of a mini- or microcomputer. The computer can do the job faster, doesn't get tired or bored, and generally doesn't make mistakes!

The overall organisation of a fusible-link PROM is very similar to that of Fig. 2.

Currently fusible-link PROMs are made with capacities up to 4096 bits. Usually they are organised externally so that the output data is in 8-bit bytes, so that the largest current device effectively stores 512 bytes. As you might expect, this type of device is intrinsically somewhat more expensive than the mask-programmed ROM, but is much more practical where single and small quantities are concerned.

Like the fusible-link PROM, the third main type of ROM device is also user programmable. But in contrast with both the fusible-link PROM and mask-programmed ROM, this third type is not confined to permanent programming. Its stored information can in fact be erased, and new information stored in its place. Hence the name erasable-PROM or "EPROM".

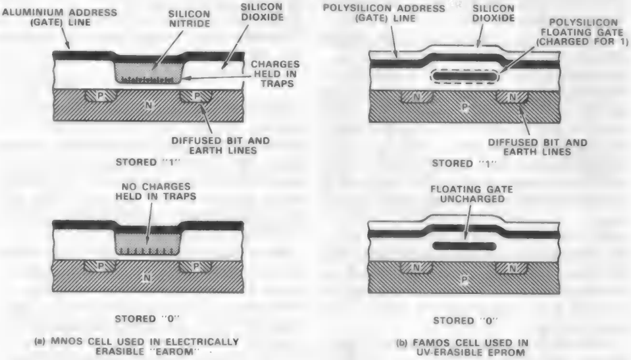
There are two different types of EPROM device in current use, the two differing in terms of the type of cell used in the storage array. Both are basically MOS devices, but one uses a metal-nitride-oxide-semiconductor or "MNOS" transistors in each memory cell, while the other uses a floating-gate avalanche-mode or "FAMOS" transistor in each memory cell.

We looked at the MNOS transistor in the last chapter, you may recall, as it is also used in the non-volatile static MOS RAM. It is basically not too different from a normal enhancement-mode MOS transistor, except that there are two dielectric layers between the semiconductor channel and the metal gate. One is the normal silicon dioxide layer, etched down to only 20 Angstroms or so, while the other is a much thicker layer of silicon nitride grown above the dioxide layer, after etching.

Where the two layers meet, crystalline defects known as "traps" are formed. These are capable of capturing charged carriers, and when trapped such charges tend to act like ■ second gate, controlling the effect of the main gate upon the transistor's channel conduction.

Fig. 4 (a) shows how such an MNOS transistor is used as an EPROM cell, to store either a "1" or a "0". The metal gate is formed by the X-address line, while the source and drain diffusions form Y-axis earth and bit sense lines respectively.

To erase the cell so that it stores a 0, the metal gate line is pulsed positively when the transistor is selected. This causes electrons to be attracted upward from the silicon substrate; they tunnel through the thin dioxide layer, and are captured by the traps. This leaves the traps filled, and substantially uncharged. In this state they have little effect upon channel conduction, and when the transistor is selected for normal readout its channel remains cut off.



(a) MNOS CELL USED IN ELECTRICALLY ERASIBLE "EAROM"

(b) FAMOS CELL USED IN UV-ERASIBLE EPROM

FIG. 4 : TYPES OF ERASIBLE PROM CELL

To store a 1 in a cell, the gate line is pulsed negative by about 25-30V when the transistor is selected. This causes electrons to be repelled from the traps in the nitride-dioxide interface, and they tunnel into the silicon substrate to leave the trap region positively charged. Due to the excellent insulating properties of the dioxide and nitride layers, this stored charge is captured almost permanently—in practice for as long as 10 years.

The effect of the stored positive charge in the traps is to assist the metal gate to produce conduction in the transistor's channel, so that when the transistor is selected for normal readout, it conducts.

As you can see, the process is reversible in that a cell may be erased and written into again, if required. And the erasure may be performed electrically, although it takes somewhat longer than the writing time: typically 10 milliseconds compared with 1 millisecond. Both are much longer than the normal read time, which is typically less than 1 microsecond.

Because of its ability to be electrically erased as well as programmed, the MNOS-cell EPROM is sometimes called the electrically-alterable ROM or "EAROM".

The second type of EPROM cell is illustrated in Fig.4 (b). Here the cell is again based on an enhancement-mode MOS transistor, but in this case there is a second polysilicon gate between the main X-address line select gate and the silicon channel. The second gate is "floating"—that is, there is no connection to it.

As with the MNOS cell, stored charge on this second gate is used to control the transistor's channel conduction. But the floating gate is charged in a different way—by inducing a controlled avalanche breakdown across the drain-channel junction, at the surface of the silicon. High-energy electrons from the avalanche breakdown are then injected into the floating gate, charging it negatively.

The avalanche breakdown of the drain-channel junction is produced by applying a high-voltage reverse bias pulse—typically around 28V, or as high as 50V with some devices. As this voltage is much higher than the normal PROM operating voltages, there is virtually no loss of stored charge from the floating gate during normal reading. The life of the stored information is thus almost indefinite—typically tens of years.

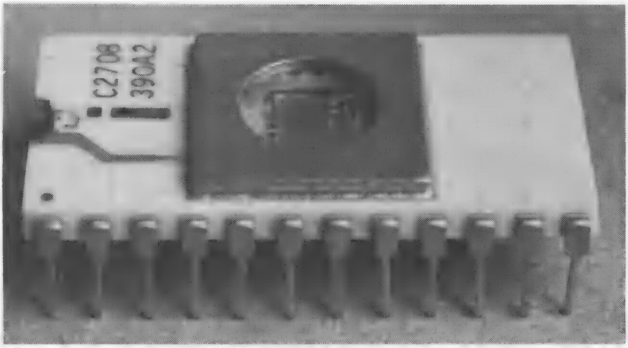
This floating-gate avalanche-mode MOS or "FAMOS" transistor was developed at the US Intel Corporation in 1971, by Dov Frohman-Bentchkowsky.

Unlike the MNOS cell, the FAMOS cell is not easy to erase electrically. The oxide layer between the floating gate and the silicon substrate is relatively thick (around 1000 Angstroms), so that the electrons trapped on the gate must be raised to a high energy level before they can escape.

Because of this the normal method of erasing an EPROM which uses FAMOS cells is to expose it to intense ultra-violet radiation. The high-energy photons impart the necessary energy to the trapped electrons, allowing them to escape to the substrate and leave the floating gate discharged. The radiation required has a wavelength of around 2537 Angstroms, and the exposure required for full erasure is about 10 watt-seconds per square centimetre.

To allow this to be done, EPROMs using FAMOS cells are fitted with a transparent quartz window immediately above the silicon chip. The device is erased by placing the window about 30mm from an ultra-violet lamp, for about 15-20 minutes. Needless to say, all cells in the device are erased simultaneously. This is in contrast with EPROMs using MNOS cells, which can be erased in individual array rows.

EPROMS which use FAMOS cells are often called UV-erasable PROMs or "UV-EPROMs", the "UV" standing for ultra-violet.



EPROM devices which use FAMOS transistor cells are fitted with a quartz window above the IC chip, to allow erasure by ultraviolet irradiation. This is an Intel 2708, with a capacity of 1024 8-bit words.

Fig. 5 shows the way both MNOS and FAMOS cells are typically connected in an EPROM storage array. As you can see the X-address lines are used to form the gate select line for array rows, while the diffused drain lines become the Y-axis bit sense lines. The overall organisation of an EPROM is basically the same as in Fig. 2.

At the time of writing, EPROMs using MNOS cells are being made with capacities of up to 4096 bits. Devices using FAMOS cells are available with somewhat larger capacity, up to 16384 bits (organised externally as 2048 8-bit bytes). Larger devices of both types are predicted before long.

Because of their ability to be programmed and reprogrammed by the user, EPROMS are particularly suited for applications requiring single custom ROMs or small quantities. They are also well suited for development work, where the final contents of a ROM may have to be arrived at by trial and error. An example is program storage for a microcomputer controller, where the program may need to be modified and/or expanded during development.

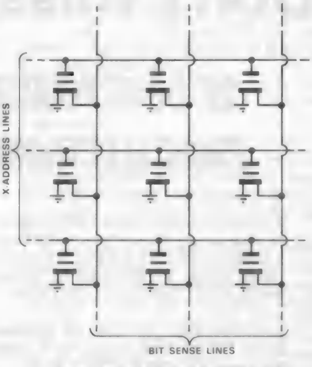


FIG. 5: ARRAY OF MNOS OR FAMOS CELLS

Once development is complete, the program or data which has been stored in the EPROM can be used to make mask-programmed ROMs if the application is one involving large quantities. This is economically desirable, as EPROMs have a somewhat higher unit cost than mask-programmed devices.

In fact to make it easy to use EPROMs for development and then change over to mask-programmed devices in production, many manufacturers make pin-for-pin equivalent devices of the same capacity, in both types. For example Intel Corporation currently makes a 16384-bit UV erasable EPROM, the 2716, and a pin-compatible mask-programmed device, the 2316E. Similarly National Semiconductor makes a 4096-bit UV erasable EPROM, the MM5204, and a pin compatible mask-programmed device, the MM5214.

Before we leave the subject of ROMs, PROMs, and EPROMS, it might be worthwhile to look briefly at the way a number of devices may be used together in the same system, to provide mores storage capacity than is available from a single device.

Basically, this is done by taking advantage of the "chip select" input involved provided on most of these devices. Two such inputs are shown in Fig. 2, for example, one effective when taken to the high logic level (CE), and the other when taken to the low logic level (CE-bar).

Like RAMs, most ROMs and PROMs are provided with at least one such input, of either type. Some have as many as four, of which some may be active-high and others active-low.

Where a ROM or PROM is provided with more than one chip enable input, they generally act together according to a logical AND function. In other words, all inputs must be taken to their active logic level, whether high or low as the case may be, before the ROM's output circuits are enabled. This can be used to simplify the decoding logic required to

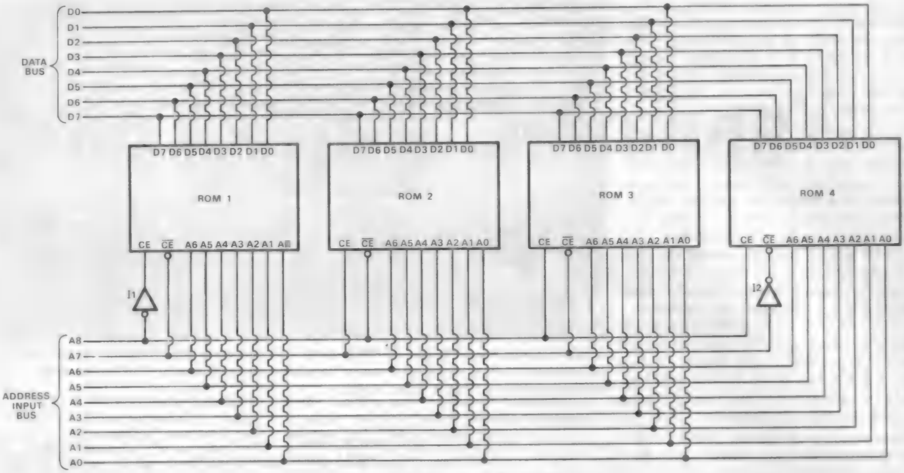


FIG. 6 USING CE AND $\overline{CE}$ INPUTS FOR MULTIPLE DEVICE DECODING

select different devices, when a number are connected into a system.

Fig. 6 shows how this is done. It shows how four of the 128 x 8 ROMs of Fig. 2 may be connected into a system, to form what is effectively a single 512 x 8 or 512-byte ROM.

As you can see, the corresponding data outputs of the individual ROMs are connected together, to form eight common data bus lines (D0-D7). Similarly the seven address inputs of the ROMs are also connected together, forming common address bus lines (A0-A6). Two further address lines A7 and A8 are added to these, to form the total of nine address lines necessary to define 512 effective memory addresses.

The two additional address lines must be effectively decoded in order to select which one of the four devices operates at any particular time. But because each ROM has both a CE and CE-bar input, the actual devices themselves perform most of the decoding internally. All that is required externally is two simple inverters, I1 and I2.

Hence by taking advantage of the two chip enable inputs on each ROM, we can arrange for ROM 1 to operate whenever A7 and A8 are both low, ROM 2 to operate when A7 is high and A8 low, ROM 3 to operate when A7 is low and A8 high, and ROM 4 to operate when A7 and A8 are both high.

ROM 1 therefore effectively provides the binary addresses from 000000000 to 001111111 (or in equivalent hexadecimal, 000 to 07F), ROM 2 the binary addresses from 010000000 to 011111111 (hex 080 to 0FF), ROM 3 the binary addresses from 100000000 to 101111111 (hex 100 to 17F), and ROM 4 the binary addresses from 110000000 to 111111111 (hex 180 to 1FF).

Incidentally, note that the technique used here is basically one of multiplexing. The ROMs are sharing a common set of data bus lines and lower address bus lines, but are effectively multiplexed onto them by means of the chip enable selection performed by the high-order address lines.

The same technique may be used with RAM devices, by the way, and also with a mixture of RAMs, ROMs and other devices fitted with chip enable facilities. We will discuss this further in later chapters.

Chapter 19

CCD's & magnetic bubbles

Two relatively new technologies are currently still emerging in the field of electronic memory devices: charge-coupled devices or "CCDs", and magnetic bubble memories. In this chapter we look at these new technologies, both of which have great potential for very high capacity "bulk memory" applications.

From a purely technical point of view, random-access memory devices of the type we have looked at in the two preceding chapters are the optimum choice where a designer has to provide a digital system with information storage. They offer the speed, elegance and efficiency of a random-access memory, combined with the reliability of a fully electronic device.

All other things being equal, therefore, the designer would prefer to use this type of memory for all of a system's memory requirements. But as it happens, all other things are not equal — costs in particular. In fact the cost of storing a bit of information in a memory device tends to be proportional to the memory's speed of operation and ease of access. Hence fast random-access memories of the type one would like to use tend to be the most expensive, while relatively slow serial-access memories tend to be much lower in cost.

For example fast semiconductor static RAMs currently cost around 1.5 cents per bit of storage, while very high capacity magnetic disc memories offer as low as .005 cents per bit.

Because of this cost structure. systems designers are generally only able to use fast random-access memory devices for part of the overall memory of a system — that part where high speed and fast access are essential. For the remainder of the memory, where speed and access are not quite as important — or less important than sheer capacity — they are forced to use lower cost serial-access devices.

Until very recently, virtually all serial-access memory devices used for such "bulk memory" applications were electromechanical, and based on either a perforated medium like paper tape or cards, or a magnetic recording medium like drums, discs (both rigid and "floppy"), or tapes. While these devices are capable of providing very high storage capacity at relatively low cost, they have a number of acknowledged shortcomings.

Being electromechanical devices with a number of moving parts, they all tend to have significantly lower reliability than purely electronic devices like semiconductor chips. They also tend to be physically bulky, to be relatively slow and to consume a relatively large amount of electrical power. All of these disadvantages have become more and more embarrassing as the electronic elements of digital systems have progressively become smaller, faster, more reliable and lower in power consumption.

Not surprisingly, a lot of effort is being spent on finding more attractive alternatives to these electromechanical devices. And two promising new technologies are currently emerging: charge-coupled device or "CCD" technology, and magnetic bubble technology.

CCD technology was developed in 1970 by Willard S. Boyle and George E. Smith, at Bell Laboratories in Murray Hill, New Jersey. It is based on devices with a metal-oxide-semiconductor structure, and is thus strictly a variant of MOS technology. But compared with the more established forms of MOS device, CCDs are much simpler in structure. This makes them potentially much easier and cheaper to make, as well as being capable of much higher element densities.

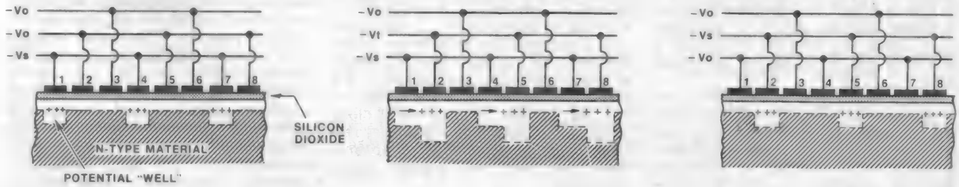
Unlike conventional MOS devices, the basic CCD involves no diffused regions in the semiconductor chip. The semiconductor chip itself consists of nothing more than a homogeneous region of doped semiconductor material, either N-type or P-type. Grown on the surface of this chip is the usual silicon dioxide passivation, with a pattern of metal electrodes deposited in turn on its outer surface.

In operation, minority carrier charges are moved through the semiconductor material, at its surface. This is done by manipulating bias voltages applied to the metal electrodes, above the oxide layer. The electrode voltages are used to create "potential wells", consisting of a localised deepening of the depletion layer formed at the semiconductor surface. The wells are then used as vehicles, to collect the minority carrier charges and move them around.

The type of CCD used as a memory device operates essentially as a long shift register. The basic construction and operation of this type of CCD are shown in Fig. 1.

As you can see, the metal electrodes deposited on the oxide layer are arranged in linear fashion. The electrodes are divided into groups of three, and each of the electrodes in each group is connected to a separate bias line. Hence electrodes 1, 4, 7 and so on are connected to one bias line, electrodes 2, 5, 8 and so on to a second line, and 3, 6, 9 and so on to a third line.

In operation, all of the electrodes are biased with a polarity which tends to repel majority carriers from the surface of the semiconductor chip, creating a depletion layer region. In the case of a



(a) STORAGE

(b) TRANSFER

(c) STORAGE

FIG. 1 : BASIC CCD CONSTRUCTION AND OPERATION

CCD using N-type material, as shown in Fig. 1, the bias voltages would all be negative (with respect to the semiconductor chip).

When the device is in the storage mode, shown in Fig. 1 (a), two of the biasing lines are held at a relatively small negative bias level Vo. This produces a uniform depletion layer over most of the chip surface. The third biasing line is taken to a somewhat higher negative voltage Vs, causing the depletion layer to deepen under the electrodes concerned. This forms ■ series of potential wells, each of which is capable of containing a charge packet of minority carriers (here they would be holes).

Note that in Fig. 1 (a) the wells under electrodes 1 and 7 are shown as containing holes, designated by the small plus signs. In contrast the well under electrode 4 is shown empty. Depending upon the logic convention in use, the "1" and "7" wells could thus be currently storing a binary 1 and the "4" well a binary 0, or vice-versa.

To move the stored information along the device, the bias line voltages are first changed as shown in Fig. 1 (b). Here bias Vs is maintained on the original storage site electrodes, but the bias on the immediately adjacent electrodes on the "forward" side is increased to a negative voltage Vt which is even higher than the voltage Vs used to maintain the storage wells. Typically it is around 10 volts.

The effect of this higher voltage is to create even deeper potential wells alongside the original wells, as shown. As a result the charges transfer into the deeper wells.

The CCD is then returned to the storage mode by reducing the bias on the original site electrodes to the threshold voltage Vo, and that on the new site electrode to the storage bias Vs. This is shown in Fig. 1 (c), and as you can see the situation is similar to that of (a) except that the charge packets have been moved one position along.

By repeating this cycle of events over and over again, the information contained in the charge packets may be moved along the CCD register. Typically this can be made to take place at clock rates of around 5MHz.

The packets of minority carriers may be introduced into the CCD device in a variety of ways. One way is by using a diffused emitter, to introduce them by injection; another way is to generate them within the chip by irradiating it with photons of visible or ultraviolet light.

At the end of the CCD register, the presence or absence of charge packets may be again detected in a variety of ways. One way is to use a diffused collector, while another is to sense them by capacitive coupling to an additional metal electrode on the oxide surface.

Although the performance obtained from planar, three-phase CCDs using a structure like that of Fig. 1 was quite promising, they were found to have a number of problems.

One problem was signal deterioration due to imperfect transfer of the charge packets from one electrode site to the next. Although even the first CCDs had quite a high transfer efficiency — around 98% — this was still too poor to make practical long shift registers.

The main reason for this turned out to be the existence of surface discontinuities in the semiconductor chip. These provided "traps", which tended to collect minority carriers from passing packets and release them into following empty wells. Another reason was the finite time taken for carriers to transfer between wells (this tends to cause transfer efficiency to droop at high clock rates).

The number of surface trap states was reduced significantly by adopting different fabrication techniques, and by increasing the purity of the silicon material. The transfer time problem was similarly improved by reducing the electrode size and spacing, so that the transfer distance was reduced. This enabled transfer efficiency to be increased quite significantly, to around 99.99%.

The other main problem with early CCDs was difficulty in fabrication. To obtain efficient transfer between electrode sites, the electrodes must be spaced very close together. The gaps between them must be no more than about 3 um (micrometres), to ensure sufficient coupling between the potential wells. Naturally enough this posed quite a problem in terms of production tolerances, to ensure that the gaps were small enough without producing short circuits.

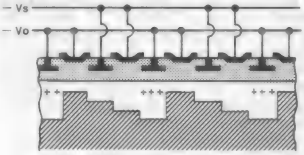


FIG. 2 : TWO-PHASE CCD WITH TWO-LAYER METALLISATION

The most successful solution yet found for this problem has been to adopt a two-layer metallisation scheme, as shown in Fig. 2. Here adjacent electrodes are at different levels, one being on the top of the oxide layer as before, and the other buried in the oxide. They can thus be arranged to overlap slightly, giving very close effective spacing of the potential well sites but without the problems associated with the original planar construction.

Actually when this two-layer metallisation scheme was tried, it was found that it offered another advantage: by interconnecting adjacent "high" and "low" electrodes as shown, the CCD could be made to work with only two-phase clock signals instead of three. This is because applying the same bias voltage to the two electrodes produces two different well depths. The buried electrode, being closer to the semiconductor chip, produces a deeper well.

Hence with only two bias voltages Vo and Vs, applied to alternate pairs of electrodes as shown, "staircases" with a total of four different potential well levels are produced. By simply alternating the voltages on the two bias lines, the staircase patterns are caused to move along the register as before.

Note, however, that the direction of motion is now a function of the device structure. This is in contrast with the planar device shown in Fig. 1, where movement could be produced in either direction at will by manipulating the three clock signals applied to the bias lines.

The CCD structure of Fig. 2 is therefore only suitable for making unidirectional shift registers, but this is not really a disadvantage for most memory applications. In any case it is far outweighted by the advantages: fabrication is easier, only two clock signals are required, and each stage of the CCD register is now only two electrodes long — giving higher packing densities.

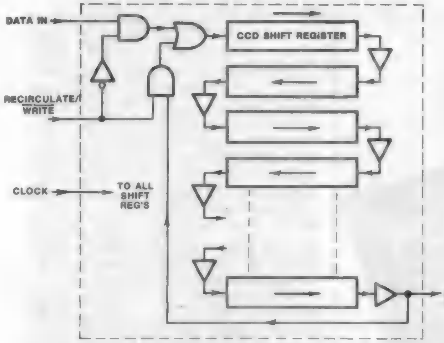
Using this type of structure designers have been able to reduce the size of a single CCD shift register stage to around 200 square micrometres, allowing memories with a capacity of 65,536 bits to be built on chips around 5mm square. Contrast this with the 4096 bits of storage currently being achieved on the same sized chip with static MOS RAMs and you can begin to see the potential of CCD technology.

It is predicted that by 1980 it will be possible to have CCD memories with a capacity of 128,000 bits on a 5mm square chip. By 1985 even this figure is likely to have quadrupled!

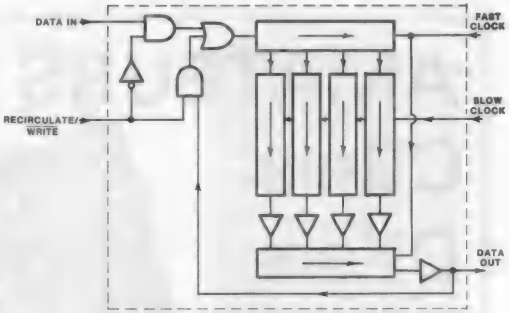
At the moment there are three broad ways in which memory CCDs are organised, and these are shown in Fig. 3. The serpentine loop scheme in (a) is the simplest, and was used on the earliest devices. As you can see it involves connection of all internal shift registers into a continuous serial register, broken only by small amplifiers used to regenerate the charge packets. The name "serpentine" comes from the way the registers are arranged on the CCD chip, so that the signal path snakes back and forth.

Logic gates are used as shown so that data may be fed serially into the memory, and then continuously recirculated around the loop. While recirculating, it may also be read out nondestructively via the data output.

The serpentine organisation gives quite good results, but because it con-

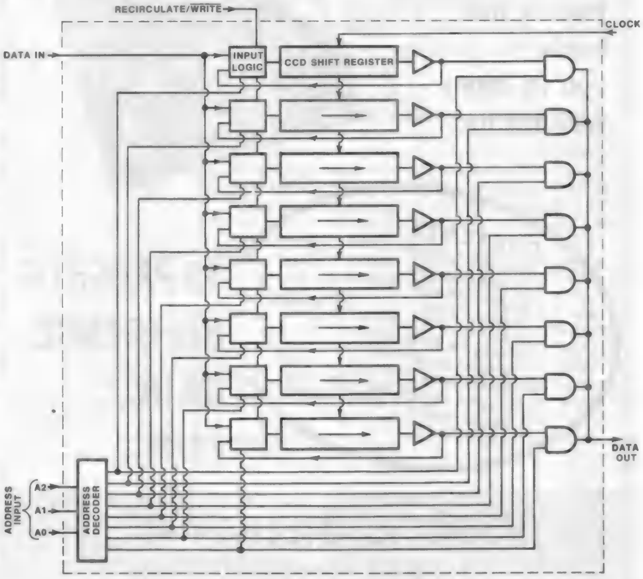


(a) SERPENTINE CCD



(b) SERIAL-PARALLEL-SERIAL CCD

FIG. ■ : CCD ORGANISATION



(c) LINE-ADDRESSED CCD "RAM"

sists basically of a single long serial register, access time tends to be fairly long for a given clock rate. Also the need for a lot of regeneration amplifiers tends to increase power dissipation.

In an effort to overcome these disadvantages the serial-parallel-serial organisation shown in (b) was developed. As the name suggests this uses a three-stage scheme. The data is read into the memory serially, at a fairly high clock rate, into an input register. It is then shifted out in parallel into an array of parallel registers, which form the main memory section of the chip. The parallel registers work at relatively low speed, and this together with the fact that they are relatively short means that only one set of regeneration amplifiers is required. At the end of the parallel registers the data is then loaded into an output register, where it may be shifted out again at a high clock rate.

The power consumption of the serial-parallel-serial CCD memory is significantly lower than that of the serpentine organisation. This is partly due to the fact that most of the memory works at relatively low speed, and partly because of the smaller number of regeneration amplifiers. The reduced number of regeneration amplifiers also means that there is more room available on the chip for actual memory storage. And because the data is stored mainly in parallel form, access time can be lower than with the serpentine approach.

The third type of CCD organisation is shown in Fig. 3(c), and is known as the line-addressed CCD "RAM". As you can see it involves a number of separate CCD shift registers. These are clocked simultaneously, but not accessed in parallel. Instead they are accessed individually and separately via logic driven by an address decoder. Any one of the registers may be written into or read from, by applying the corresponding address code to the decoder address inputs.

Normally the registers are all recirculating individually. To write into ■ register, its address code is applied to the decoder inputs, and the recirculate/write logic input taken to its write enable level (here logic low). Serial data applied to the memory input is then shifted into the selected register. When the data ends, the recirculate/write logic input is taken back to its original logic level to restore the register to the recirculate mode.

To read ■ particular register, all that is necessary is to apply its address code to the decoder inputs. The data in the register will then be available at the data output of the device, as it recirculates. Again this read operation is non-destructive, as the recirculated data is not affected.

Although it is obviously not a true random-access memory, the line-addressed CCD does offer some of the advantages of a RAM. Because the data is stored in a number of individually addressable registers, it is more readily accessible than with either the serpentine or serial-parallel-serial schemes. And because the individual lines are relatively short, few regeneration amplifiers are needed, keeping power dissipation down.

In short, the line-addressed CCD "RAM" is the one that offers the highest performance, and the one which seems to have the greatest potential in terms of future development as a low-cost, high reliability replacement for electromechanical bulk memory devices.

At the time of writing, Fairchild Semiconductor has just released a line-addressed CCD memory device with a capacity of 65,536 bits. It is organised as 16 registers, each of 4096 bits, and may be clocked at rates between 1MHz and 5MHz. At 5MHz clock rate, the average access time or "latency" is 410us (microseconds), which compares very well with electromechanical bulk memories.

However there is one aspect of performance in which CCDs do not compare favourably with electromechanical bulk memory. They are volatile — i.e., the stored information is lost if power is removed from the memory chip. This is in contrast with virtually all electromagnetic memories, where the information remains stored in either mechanical or magnetic form.

Of course this disadvantage may be overcome by using a battery-backup power supply, so that power is never removed from the CCD memory chips. However this can be expensive where large memories are concerned. Needless to say, designers would prefer having solid state bulk memory devices which were themselves non-volatile, to obviate this problem.

Happily this disadvantage does not apply with the second emerging memory technology — magnetic bubbles. As the name suggests these store information in magnetic form, and can be arranged to provide non-volatile storage quite easily.

Like CCD technology, magnetic bubbles were developed at Bell Laboratories. The technology was first announced in late 1967, but development has been rather slow since then, and it is only now in 1977 that the first commercially available magnetic bubble memories have emerged.

Unlike CCDs, magnetic bubble memories are not semiconductor devices at all. Many of the steps used to manufacture them are similar to those used to manufacture CCDs and other integrated circuits, and like ICs they are relatively small devices with no physically moving parts. But there the similarity ends. In fact a bubble memory is closer in operation to a magnetic disc or drum, in that it stores information in the form of a magnetic recording.

The only difference is that in a magnetic disc or drum, the magnetic medium must be rotated in order to store the recording over its surface. In the bubble memory, the medium stays fixed, while the actual "recording" is made to move around!

The operation of a bubble memory depends upon the magnetic properties of a class of materials known as orthoferrites. In particular the material most used to date has been a magnetic garnet, in single crystal form. In this form, the properties of this type of material are such that it may be magnetised much more readily in one crystalline axis than in the others.

If a very thin slice of the material is produced, with this axis of preferred magnetisation perpendicular to the plane of the slice, it thus tends to be magnetised in either the "upward" or "downward" directions. In fact even when the slice as a whole has no net magnetisation, there tend to be localised regions or "domains" within it which are magnetised one way or the other. All that happens when there is no nett magnetisation is that the domains magnetised in one direction are balanced out by those magnetised in the other.

An interesting thing happens in such a slice if an external magnetic field is applied, again perpendicular to the wafer. The domains which are orientated in the opposite direction to the field tend to contract, while those orientated in the same direction as the field expand. By increasing the field intensity up to a certain point, the oppositely magnetised domains can be made to shrink into tiny "bubbles", as small as 2 micrometres in diameter. The other domains expand out and merge to occupy all of the remainder of the slice.

Actually it is possible to create new bubble domains in the slice, by passing current through a small loop of conductor close to the surface, to produce a small localised field opposing the main field. It also proves to be possible to manipulate the bubbles present in a slice, so that they can be used to store information. And the bubbles will continue to exist in the slice as long as required, provided that the main "bias" field is maintained.

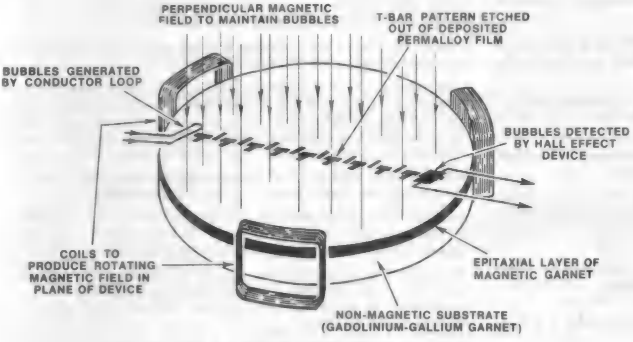

PERPENDICULAR MAGNETIC FIELD TO MAINTAIN BUBBLES
T-BAR PATTERN ETCHED OUT OF DEPOSITED PERMALLOY FILM
BUBBLES GENERATED BY CONDUCTOR LOOP
BUBBLES DETECTED BY HALL EFFECT DEVICE
COILS TO PRODUCE ROTATING MAGNETIC FIELD IN PLANE OF DEVICE
EPITAXIAL LAYER OF MAGNETIC GARNET
NON-MAGNETIC SUBSTRATE (GADOLINIUM-GALLIUM GARNET)

FIG. 4 : BASIC MAGNETIC BUBBLE DEVICE

Fig. 4 shows the basic structure of a practical bubble memory device. The heart of the structure is a very thin layer of magnetic garnet, grown epitaxially on the surface of a thicker slice of non-magnetic rare earth (gadolinium-gallium) garnet which serves as a substrate. A permanent magnet is used tc apply the perpendicular bias field required to maintain the bubbles.

Bubbles are produced at one end of the device by a small conductor loop near the surface. This acts as the input of the device. Once created, the bubbles are manipulated and moved around, by means of a further magnetic field which rotates in the plane of the slice. The rotating field is produced by a set of coils, as shown, and acts on the bubbles via a pattern of thin and magnetically "soft" permalloy pole-pieces on the surface of the chip. The pole-pieces are extremely tiny, being produced by depositing a layer of permalloy on the chip and then photo-etching it using IC fabrication techniques.

Typically the pole-pieces are shaped either like arrowheads, or as alternate "bars" and "tees" as shown. The latter arrangement is known as the "T-bar" system. Either way, tne pole-pieces are arranged as a long continuous chain, which may be laid out in serpentine fashion.

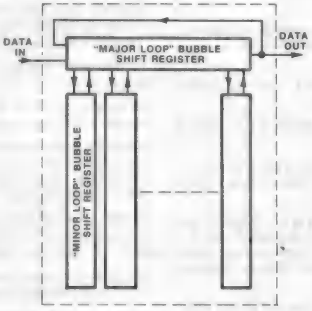

DATA IN
"MAJOR LOOP" BUBBLE SHIFT REGISTER
DATA OUT
"MINOR LOOP" BUBBLE SHIFT REGISTER

FIG. 5 : TYPICAL BUBBLE MEMORY ORGANISATION

What happens is that the tiny pole-pieces become dynamic induced magnets, under the influence of the rotating magnetic field produced by the coils. And the shape of the pole-pieces together with their spacing causes any bubbles present to be drawn along the chain. Each full rotation of the field causes a bubble to move one stage along the pattern — i.e., one T-and-bar in the case of a T-bar device.

The pattern produced by the pole-pieces thus becomes a shift register, capable of storing information carried by the bubbles. The register clocking rate corresponds to the rotational frequency of the field produced by the coils. Currently this is around 1.25MHz.

At the end of the register, the presence or absence of bubbles is detected by a suitable magnetic sensor. Typically this is a small Hall-effect element, as shown in Fig. 4. The readout may be either destructive or non-destructive, as desired, so that recirculating registers are quite feasible.

Many of the early bubble memories used a single long register of this type, arranged in serpentine fashion rather like the CCD register organisation of Fig. 3(a). Devices of this type are practical, and have been used for no-moving-parts data recorders and similar applications. However until the fabrication techniques used to make bubble devices have been fully refined, this type of construction tends to have yield problems. Any slight fault anywhere in the register can render it unusable.

More recent bubble memories have tended to use the organisation shown in Fig. 5, known as the "major-minor loop" system. Here the so-called "major" loop register is used to feed serial information into and out of the device, while the actual storage is performed by a number of "minor" loops. Each of the minor loops is of the recirculating type.

A typical device of this type has some 157 minor loops, each consisting of 641 stages. This provides a potential storage capacity of 100,637 bits, with an average access time of around four milliseconds. Actually the idea of having 157 minor loops is that there is some redundancy, to allow for faulty loops. The device is nominally regarded as having 144 loops, giving a nominal capacity of 92,304 bits. The additional 13 minor loops are to ensure that this capacity will be available.

This device is housed in a modified 14-pin dual in-line package which measures 25 x 27 x 10mm. Included in the package are the bubble chip, a pair of permanent magnets for the bias field, the coils for the rotating field, and a magnetic shield to protect the device from external fields.

Eight such memory devices, together with the necessary interfacing logic, can be mounted on a single printed circuit board to form a 92,304-byte bulk memory weighing less than 320 grams.

The future for bubble technology seems bright. Already devices with a capacity of 250,000 bits on a single chip have been produced in research laboratories. By 1980 it is predicted that bubble sizes will be down to below 1um, allowing memory densities of around 150,000 bits per square millimetre. It should then be possible to produce single chips with capacities of around 10 million bits, at a cost which should make them very attractive as replacements for floppy discs and similar devices.

Glossary of Terms

ADDER: A circuit or device used to perform addition, usually binary addition. There are two broad types, the serial or bit-by-bit adder and the parallel or all-bits-together adder.

ADDRESS: A binary number specifying a particular storage location in a memory system. The number of bits in the address (N) is directly related to the number of addressable locations (A): $A=2^N$.

ADDRESS BUS: A group of logic signal lines along which are propagated the various address bits required for specifying any of the storage locations in a memory system.

ADDRESS LINE: Within a memory device, one of the signal lines used to gain access to the storage cells. More generally, any logic signal line carrying address information.

ALU: Contraction of Arithmetic Logic Unit, a digital sub-system capable of performing a variety of binary arithmetic and logic functions in response to logical command signals.

ASCII: Contraction of American Standard Code for Information Interchange, a commonly used code for alphanumerical information.

BASE: The base or radix of a numbering system is the number of values which may be taken by a digit. With conventional fixed-weighting notation, the weighting of the digits also corresponds to integral powers of the base, the powers normally increasing integrally from right to left.

BCD CODES: Binary-coded-decimal codes, which use groups of four binary digits or bits to represent decimal digits. Both fixed and variable weighting codes are possible; there are over 7 million possible BCD codes.

BINARY NOTATION: The system of numbering using a base of 2. Each digit has only two possible values: 1 or 0.

BIT: Contraction of "Binary digit", or a digit in the binary notation system. The smallest possible amount of information, having only two possible values: 1 or 0, truth or falsity.

BOUNCE: The characteristic of mechanical switching contacts, wherein they tend to perform a number of very rapid transitions whenever they are nominally opening or closing.

BUSLINE: A data path used in a digital system to link a number of different sources and destinations, by means of multiplexing.

BYTE: Commonly, a group of 8 bits. Strictly, any group of bits handled together.

CAM: Content-addressable memory. A static RAM device provided with additional logic so that it can search through its stored data seeking a match with an externally presented data word or "key".

CCD: Charge-coupled device. A semiconductor device in which data may be stored as tiny lumps or "packets" of electrical charge, manipulated in the surface layer of the semiconductor by means of potentials applied to metal electrodes above the oxide passivation. Used mainly for serial-access memories, long shift registers and optical image processing.

CLOCK: The source of time reference signals used by a digital system to ensure that the various parts of the system work together correctly.

COMPLEMENT, LOGICAL: The logical complement of a term is another term which is always false when the first term is true, and always true when the first term is false. That is, its logical opposite.

COMPLEMENT, NUMERICAL: The numerical complement of a number is another number which when added to the first number, produces a sum of zero. That is, the negative equivalent of the number.

COUNTER: A device or circuit which is capable of counting the number of pulses fed to it, using a suitable number system.

DATA BUS: A group of logic signal lines along which are propagated the various bits of data words involved in a digital system.

DECADE COUNTER: A counter designed to count 10 pulses before resetting, and whose count is capable of being displayed in decimal form.

DECODER: A digital circuit designed to accept information in coded form, and produce corresponding uncoded information.

DELAY ELEMENT: A digital element designed to effectively delay the propagation of a pulse or level transition, so that it arrives at its destination later than otherwise.

DEMULTIPLEXER: A digital circuit designed to separate the various signals or pieces of information which have been multiplexed together through a common channel.

DISTRIBUTOR: A digital circuit designed to generate a set of sequential timing signals, capable of being used to initiate a sequence of events.

DYNAMIC RAM: A random-access read-write memory device in which the individual storage cells require regular "refreshing" in order to retain the stored data. In most cases the storage cells consist primarily of a single capacitor, with the data stored as the presence or absence of charge.

ENCODER: A digital circuit designed to accept information in uncoded form, and generate corresponding coded data.

EPROM: A programmable read-only memory device which allows the stored data to be "erased" or destroyed, and other data stored instead. The most common type of EPROM is the UV-erasible type, in which the data is erased by irradiating the semiconductor chip with intense ultra-violet radiation. To allow this the device package is fitted with a transparent quartz window.

FAMOS TRANSISTOR: Floating-gate avalanche mode MOS transistor, used as the data storage element in most EPROM devices.

FAN-IN: The electrical loading effect associated with a device input, expressed either in terms of actual electrical quantities (current, resistance, capacitance) or as a multiple of a defined "standard input".

FAN-OUT: The electrical loading capability of a device output, expressed either in terms of actual electrical quantities (current, resistance, capacitance), or as the number of defined "standard inputs" which it is capable of driving reliably.

FIFO: First-in first-out memory device. A static RAM device provided with additional logic which makes it behave like a serial memory device or "flexible shift register". Data may be written into the device and read out once at different rates, simultaneously, but in the same sequence. Used to provide "buffering" or coupling between sections of a digital system operating at different data propagation rates.

FIRMWARE: Instructions forming a program for a computer or controller, stored as data in a ROM or PROM memory device.

FLIPFLOP: A digital element capable of occupying either of two stable states, and therefore capable of storing one bit of information.

FUSIBLE-LINK PROM: A programmable read-only memory in which the storage cells are basically small metal links which are programmed by fusing or "burning" them away with high amplitude current pulses. This type of PROM is not erasible.

GATE: A digital logic element whose output logic value is related to the values presented at its inputs by one of the based logical functions — AND, OR, NOT, NAND or NOR.

GLITCH: A spurious and potentially troublesome pulse, usually narrow, caused by timing errors within a digital system.

HEXADECIMAL NOTATION: A numbering system using a base or radix of 16. To represent the 16 digit values, the first six alphabet characters are used to augment the decimal number symbols 0-9.

JOHNSON COUNTER: A counter of the twisted-ring type, in which one end of a shift register is connected to the other via an inverter to form a ring. Capable of counting to 2N, where N is the number of flipflops in the ring.

KARNAUGH MAP: A topological diagram used to aid in minimalisation of complex logic functions. A grid in which each cell corresponds to a different truth-value combination of the terms involved in the function.

LOGIC CONVENTION: The assigned relationship between electrical values and logical truth values in a digital system. An example is positive logic, where logical truth (1) is represented by the more positive voltage level, and logical falsity (0) by the more negative voltage level. The opposite convention, negative logic, is equally valid.

MAGNETIC BUBBLE: A tiny magnetised region or domain which can exist as an entity within an orthomagnetic material such as magnetic garnet. The bubbles may be manipulated within the garnet by means of external magnetic fields and metal electrodes on the garnet surface, and may thus be used for data storage.

MASK-PROGRAMMED ROM: A read-only memory device in which the data is stored permanently when the device itself is made, using the fabrication masks. Used where a large number of ROMs are required to contain the same information.

MEMORY CELL: An element within the storage array of a memory device which is capable of storing one bit of data.

MICROPROCESSOR: An integrated circuit (IC) performing all or most of the functions of the central processor unit (CPU) section of a digital computer.

MODULO FACTOR: The number of counts performed by a counter circuit before it returns to its original or reset state.

MULTIPLE PRECISION: The technique of performing binary arithmetic operations on large numbers by splitting them into smaller sections, and operating on the sections separately. Allows circuitry designed to handle small numbers to deal with much larger numbers, effectively multiplying the precision attainable.

MULTIPLEXER: A digital circuit which processes a number of separate signals to enable them to be passed through a single data channel.

MULTIPLEXING: The technique by which a number of separate signals are interleaved so that they may be passed through a single data channel, and then separated out again. In digital systems, the interleaving is generally performed by cyclic sequential sampling — i.e., time multiplexing.

OCTAL NOTATION: A number system using 8 as its base or radix, and the decimal numerals 0-7 inclusive to represent the 8 possible values of each digit.

ORGANISATION: The way in which the data bits stored in a memory device or system are effectively grouped, from an external functional point of view. Usually expressed in terms of the number of words of a certain bit length. Hence a memory with 8,192 bits of storage may be described as organised into 1024 words of 8 bits, or "1024 x 8".

PROCESSOR: A digital subsystem which performs data processing, including arithmetic and logic functions. Usually incorporates an arithmetic-logic unit (ALU), together with various registers and timing circuitry. A processor forms one of the main elements in a computer.

PROGRAMMABLE LOGIC ARRAY: A read-only memory arranged so that it may be used to synthesise a required complex logic function, or a number of such functions. The ROM address inputs become the logic term inputs while the data outputs become the functions outputs.

PROM: A programmable read-only memory, i.e., one which allows storage of data either permanently or semi-permanently subsequent to manufacture.

PROPAGATION DELAY: The delay introduced by a digital circuit element as a result of its internal circuitry; the time taken for a change in its input signals to be reflected by a change at its output.

RADIX: The base of a numbering system.

RAM: Random-access memory, normally read-write memory. A data storage device or system in which all data storage locations are equally accessible for either storage (writing) or retrieval (reading).

READING: The operation whereby a memory device is arranged to present on its output lines a replica of the data stored in the location specified by the address inputs. In most memory devices the reading is performed nondestructively — i.e., the stored data itself is not disturbed.

REGISTER: A device used to store or manipulate numbers or other data; usually a group of flipflops.

RING COUNTER: A counter formed by connecting one end of a shift register to the other. Counting takes place by moving a pattern of bits around the resulting ring.

RIPPLE-CARRY COUNTER: A counter in which each new count must propagate through each element in turn before it is correctly registered. Until the propagation is complete, the counter outputs may pass through a number of incorrect transition states, causing "glitches".

ROM: Read-only memory. Normally a random access memory device in which the stored date is effectively permanent in normal operation and can only be read out, not overwritten.

SCALER: A device or circuit designed to deliver an output pulse after a specific number of input pulses, repetitively if necessary.

SCHMITT TRIGGER: A controlled switching circuit whose threshold for rising inputs differs from that for falling inputs, so that it exhibits hysteresis.

SERIAL-ACCESS MEMORY: A memory device or system in which the data is stored sequentially in time, and can only be retrieved in the order in which it was stored.

SHIFT REGISTER: A digital circuit capable of moving information in serial or bit-by-bit fashion between storage elements.

SOFTWARE: Sequences of instructions which form programs for a computer or controller, and which are generally stored like data in RAM memory devices within the computer system. The prefix "soft" is meant to differentiate them from hardware, i.e., physical devices.

STATIC RAM: A random-access read-write memory device in which the data storage cells retain the stored data while ever the power is applied, unless new data is deliberately stored in its place. The stored data does not require periodic "refreshing".

SYNCHRONISER: A digital circuit which accepts random logic signals, and generates corresponding signals which are synchronised with the clock pulses of a system.

SYNCHRONOUS COUNTER: A counter whose elements are arranged so that all changes of state occur in synchronism.

TRUTH TABLE: A table showing the logical function of a digital element or circuit, in terms of the output truth values corresponding to all possible truth value combinations for the inputs.

TWO'S COMPLEMENT NOTATION: A variant of binary notation, in which the most significant bit is used to indicate sign and negative numbers are produced by complementing individual bits, and then adding 1 to the result.

VOLATILITY: That characteristic of a memory device which results in its stored data being lost if the power is removed. A memory device which does not lose its stored data when the power is removed is described as non volatile.

WRITING: The operation of storing data in a particular location within a memory device or system, the location being specified by the address word applied to the memory address lines at the same time. The storage operation is generally destructive — i.e., any existing data stored in the location concerned is automatically destroyed.

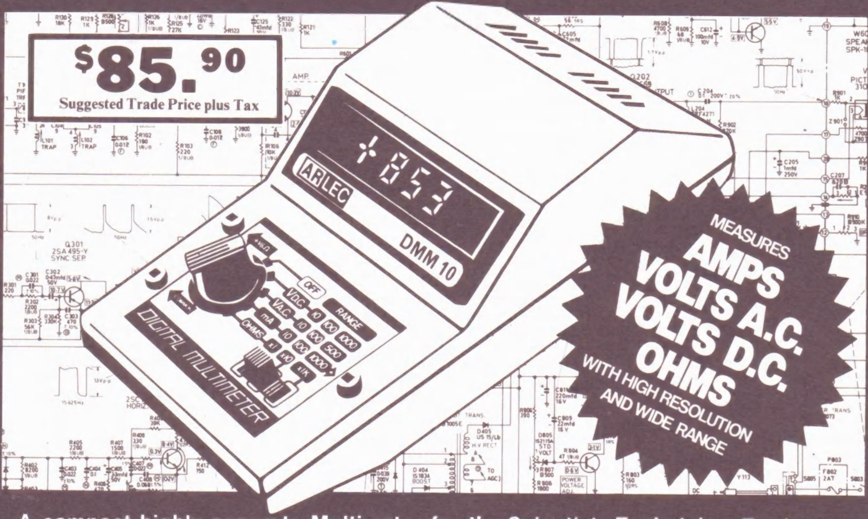
WORD: A group of binary digits or bits, which are handled together as an information unit. Some digital systems manipulate data in fixed word lengths throughout, while others may use a number of word lengths for different purposes. Thus a computer system may use 8-bit words for numerical data and instructions, and 16-bit words for addresses.

Index

Advance into the new era of precise electronic measurement

ARLEC **DMM 10**

DIGITAL MULTIMETER



$85.90
Suggested Trade Price plus Tax

MEASURES
AMPS
VOLTS A.C.
VOLTS D.C.
OHMS
WITH HIGH RESOLUTION
AND WIDE RANGE

A compact highly accurate Multimeter for the Scientist, Technician, Tradesman, Electrician and Advanced Hobbyist. ● SIMPLE 2 SWITCH OPERATION ● EASY READ DIGITAL DISPLAY ● 1% ACCURACY

Features:

- Large ⅜" (9.5mm) LED Numerals
- Automatic Decimal Point
- 3 Digit 0-999 Display
- Instantaneous Non-flashing Readout
- Zero Locked
- 7 MEGOHM Input Impedance.
- 12 Separate Measuring Ranges
- Over-range Indication
- Overload Protected
- 1% Accuracy

Supplied complete with:

- RECHARGEABLE NI CAD BATTERIES
- SEC APPROVED PLUG-IN ADAPTOR/CHARGER
- CONNECTOR LEAD FOR CHARGING FROM AUTO BATTERY
- TEST PRODS
- OPERATING INSTRUCTIONS
- 12 MONTH GUARANTEE

DESIGNED & MANUFACTURED IN AUSTRALIA BY

SEE THEM AT YOUR LOCAL ELECTRONICS STORE OR THE A+R ELECTRONICS BRANCH IN YOUR STATE.

ARLEC
QUALITY PRODUCTS

A+R Electronics
A MEMBER OF THE A+R-SOANAR ELECTRONICS GROUP
30 Lexton Road, Box Hill, Vic., 3128, Australia.

VICTORIA: 89 0661
N.S.W.: 78 0281
STH. AUST: 51 6981
QUEENSLAND: 52 5421
WEST AUST: 81 5500
HONG KONG: (3) 89 1271
TOKYO: 585 8025

THE HEATHKIT H11 DIGITAL COMPUTER

Two of the finest names in modern electronics, Heath and Digital Equipment Corporation (DEC) combine to bring you the world's first 16-bit computer priced within reach of the general public!

HEATH
Schlumberger

Heath and DEC join forces to bring you mini-computer performance at a microcomputer price! The H11 features a fully wired and tested DEC KD11F board that contains the 16-bit LSI-11 CPU, 4096 x 16 read/write MOS semi-conductor memory, DMA operation; and includes the powerful PDP-11/40 instruction set, PLUS Heath/DEC PDP-11 software. Equivalent commercial versions of the H11 would cost $1,000's of dollars more!

The new Heath/DEC H11 personal computer is one of the most powerful and sophisticated units available today! It combines the advanced, performance-proven hardware and software of the LSI-11 with Heath's expertise in kit design and documentation to bring you a personal computer of almost incredible power and flexibility. Equivalent commercial versions of the H11 would cost twice as much, and you still wouldn't get the superior documentation and support of the H11!

POWERFUL HEATH/DEC PDP-11 SOFTWARE AT NO EXTRA COST!

The H11 includes a sophisticated software system that lets you get your computer up and running with practical programming capabilities. This paper tape based software would cost over $1200 if purchased separately. A minimum of 8K memory is required to run the software. The programs include:

ED-11. Assists you in the creation and modification of ASCII source tapes, also used to write assembly language programs and for general text editing or word processing functions.

PAL-11S. Relocatable assembler converts ASCII source tapes into relocatable binary modules. This lets you create programs in small, modular segments for easier coding and debugging. These binary modules serve as inputs to LINK-11S

LINK-11S. Link editor which links the modules created by the PAL-11S into a load module ready for execution on the H-11. The module is loaded into the H-11 via the Absolute Loader.

Absolute Loader. Loads absolute binary tapes into the H11 memory for execution.

ODT-11X. Lets you debug the programs which you have created. Permits modifying and controlling program execution "on the fly" for quick, efficient debugging.

IOX. Executive program permits I/O programming without developing device-driving programs. Links to your programs using the LINK-11S. For use with high speed paper tape reader/punch and line printer.

DUMP-AB AND DUMP-R. Lets you dump absolute binary contents of memory into the paper tape punch.

BASIC. DEC's powerful version of standard Dartmouth BASIC interpreter uses english-type statements and mathematical symbols to perform operations. Immediately translates, stores and executes the program. Includes string capability.

FOCAL™. DEC's own interpretive computer language which combines simplicity with computing power. Ideal for most scientific, engineering and math applications. FOCAL™ programs can be written and executed easily. Both 4K and 8K versions are included.

NOTE: H11 owners are eligible for membership in the Digital Equipment Computer User's Society (DECUS). This organization provides useful symposia, newsletters, program library and other useful information to help you get the most from your LSI-11 computer.

WF WARBURTON FRANKI

• ADELAIDE 356-7333 • BRISBANE 52-7255 • HOBART 23-1841
• MELBOURNE 699-4999 • PERTH 277-7000 • SYDNEY 648-1711
• AUCKLAND N.Z. 360-154 • WELLINGTON N.Z. 698-272

WF 560/77